(1)

PROCEEDINGS OF THE THIRD ANNUAL Ada SOFTWARE ENGINEERING EDUCATION AND TRAINING SYMPOSIUM

Sponsored by:

Ada Software Engineering Education and Training Team
Ada Joint Program Office

Clarion Airport Hotel
Denver, Colorado
June 14-16, 1988

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

ADA197239

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Proceedings of the Third Annual Ada Software Engineering Education & Training Symposium		5. TYPE OF REPORT & PERIOD COVERED 14 June '88 to 16 June '88
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Ada Software Engineering Education and Training Team (ASEET) Ada Joint Program Office AJPO		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS ASEET Team		12. REPORT DATE June 1988
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081		13. Number of Pages 298
14. MONITORING AGENCY NAME & ADDRESS*(If different from Controlling Office)* AJPO		15. SECURITY CLASS *(of this report)* UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A
16. DISTRIBUTION STATEMENT *(of this Report)* Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20. If different from Report)* UNCLASSIFIED		
18. SUPPLEMENTARY NOTES		
19. KEYWORDS *(Continue on reverse side if necessary and identify by block number)* Ada Programming Language; Education, Training, Abstracts		
20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)* The information contained in this publication are the abstracts from the speakers at the Third Symposium The information covers the uses of Ada in the academic world, industry, and the Armed Services.		

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ASEET TEAM MEMBERSHIP

Mr. John Bailey
Ms. Wanda B. Barber
Captain Roger Beauman
Captain Eugene Bingue
Captain Will Bralick
Captain David Cook
Captain Nancy Crowley
1LT Anthony R. Dominice
CDR Dave Endicott
Major Charles B. Engle, Jr.
Mr. Alex Grindlay
LtCol Rick Gross
Captain Jay Hatch
Captain Steven T. Holste, USMC
Paul J. Howe
Major Allan Kopp
Major Pat Lawlis
Ms. Cathy McDonald
LCDR Lindy Moran
Ms. Sue O'Neill
Mr. Jeff Riplinger
Major Randall B. Saylor
Capt. David Umphress
Mr. James J. Valentino

This Page Left Blank Intentionally

S. 1472

TABLE OF CONTENTS

TABLE OF CONTENTS
(continued)

[illegible]

High [illegible],
[illegible] Ada [illegible]
education Training [illegible]

Message From The ASEET Chair

Welcome to the Third Annual ASEET Symposium. This year we have endeavored to provide a wide range of speakers and papers from industry, academia, and government. Interaction and the exchange of ideas are paramount to the successful instantiation of Ada in these domains. This symposium provides an excellent opportunity to accomplish this goal. Our exhibitors bring you education and training materials/software tools from the leading edge of their fields. Please take the time to find out what they have to offer. There are critique forms throughout the symposium. Please fill these out for they provide the only medium through which we can improve our symposia. Thank you.

This Page Left Blank Intentionally

A SOFTWARE ENGINEERING METHODOLOGY THAT INCORPORATES CLASSROOM AND ON-THE-JOB ADA TRAINING

J.L. Graffo, Senior Software Engineer, Harris GISD
Software Operation, Melbourne, Florida

D.O. Nickens, Site Manager, Harris Data Services
Corporation, Alexandria, Virginia

T.C. Freeberg, Senior Software Engineer, Harris GISD
Software Operation, Melbourne, Florida

ABSTRACT

Few changes in software development technology have resulted in the turmoil caused by the introduction (and mandated use for DoD programs) of the Ada programming language. From its inception, issues concerning Ada development have produced diverse reactions ranging from "Ada - a total solution" to "Ada-an expensive impediment". Opinions are defended with a fervor seldom seen in the software development and engineering field. Given the misinformation resulting from these reactions, it is necessary to mitigate the impacts of using this new technology. Regardless of personal or corporate opinions, using Ada requires the implementation of a strict development methodology and changes the currently accepted approaches to software engineering. Due to these changes, training provided for both technical developers and management personnel must also be significantly altered. A solution is a software development methodology that allows for the inexperience of most available Ada developers and managers by integrating formal classroom and on-the-job training. This paper describes the key aspects of this methodology and how it is used to solve the training dilemma caused by the introduction of this new software technology.

INTRODUCTION

Proponents of Ada claim that it is ...more than just another programming language. Experiences at various corporations, where millions lines of Ada code have been developed and delivered, show that this is both a true statement and a false one.

Ada can be just another tool with which complex programming problems can be implemented. It can provide all of the same capabilities and functionality of most other programming languages. In this way it is not much different than FORTRAN or Pascal - just another language. The use of methodologies that Ada can take full advantage of, such as object-oriented design, drastically changes the focus of software development programs. The terms used to describe software systems currently (functions, for instance) are not really compatible with the terms required to make the most effective use of Ada (objects). Software development organizations must modify the methodologies used from strategies such as structured analysis and design to employ more user-friendly, Ada-compatible strategies such as object-oriented requirements and design.

Because of the immaturity of these new strategies, Ada and methodology training is crucial to the success of an Ada program. Problems normally associated with software engineering training have increased in number and have been magnified due to the new technologies that Ada supports. This is a prime concern to functional and program managers for several reasons. One, it is very difficult to find and keep top-notch software developers. This is not just an Ada problem, but is exacerbated because many top notch developers have not embraced the Ada programming language. Two, many current Ada developers are recent college graduates with very limited experience. Three, current developers with Ada experience tend to be at the programmer level rather than the requirements or design level. And more than most other languages, there is a very big difference between programming in Ada and designing for Ada. Four, few experienced Ada programmers or designers have the experience required tomanage a large project.

Formal classroom training provided to Ada trainees consists of a 40 hour course, spread over eight or more weeks, with required homework and in-class problem solving. At the end of the course, students are expected to be able to program in Ada in a manner that can be best described as immature. (It is surprising how similar Ada and FORTRAN can appear in the hands of a novice!) Students should be familiar with advanced Ada concepts, such as data abstraction and information hiding, but they may not yet have the ability to design these concepts into real systems.

On-the-job training is an integral part of the software engineering methodology. By recognizing the need for this training in advance and by carefully tailoring the software engineering methodology to allow for it, maximum benefits and minimized risks are realized. On-the-job training is used to rapidly mature the skills that were acquired in the classroom. The remainder of this paper will describe the key aspects of a software engineering methodology tailored for training and how this training has helped to solve the problems associated with the introduction of the Ada technology. This methodology has been used successfully to produce over 150,000 lines of Ada on a fixed-cost DoD contract.

METHODOLOGY

The decision to use a particular software development methodology for a company's first large Ada program is a very difficult one. In many cases, those charged with making policy decisions within a company have heard or read that object-oriented approaches yield a more maintainable and user-oriented system. But due to the lack of local or personal experience, the decision-makers are reluctant to stray far from the techniques and methodologies that they understand.

The authors of this paper had very similar experiences on their first Ada project. It is important to note that their first Ada project was a fixed cost contract to deliver over 100,000 lines of Ada within very tight schedule constraints. In light of this, the decision was made to use an analysis and design methodology that was already understood by the software development organization - Structured Analysis and Design. But, it was also decided that Object-Oriented Design would be used as soon as it was fully understood and the transition from Structured Analysis and Design to Object-Oriented Design was fully defined and documented.

The authors believe that once more precise and rigorous methods are defined for performing object-oriented requirements analysis, an overall object-oriented approach to Ada software development will be desireable. The authors also believe that in light of this current deficiency in object-oriented methodologies, a well defined and understood methodology and a strong software organizational structure are more likely to positively affect an Ada development than an object-oriented methodology that is less understood. Therefore, the thrust of the methodology discussed in this paper is: (1) to get personnel on-board quickly, (2) put an organization in place that encourages creativity, provides open communication, and promotes job understanding, (3) teach the organization members the Ada language and the VAX Ada environment, (4) teach them the Structured Analysis and Design techniques and Object-Oriented Design, and (5) standardize the use of Ada, the products of each phase, and the tailoring of DoD-STD-2167.

Project Organization

One of the the most important aspects of software development needed to successfully complete an Ada project is the project or program organization. It is vital to the success of an Ada project to have an organization in place in which each member fully understands their responsibilities.

The organization put in place to perform the project is shown as Figure 1. The functional manager and the program manager were assigned to handle the hiring and handling of personnel and related issues and to be responsible for the program and accountable to the customer, respectively.

The functional manager's responsibilities included:

a. hiring of program personnel
b. review of personnel performance
c. assigning personnel to various program areas
d. tracking the program at a detailed level
e. internal meetings and reviews

The program manager's responsibilities included:

a. interfacing and negotiating with the customer
b. maximizing program profits
c. maintaining the program schedule
d. allocating funding to various program areas
e. customer meetings and reviews

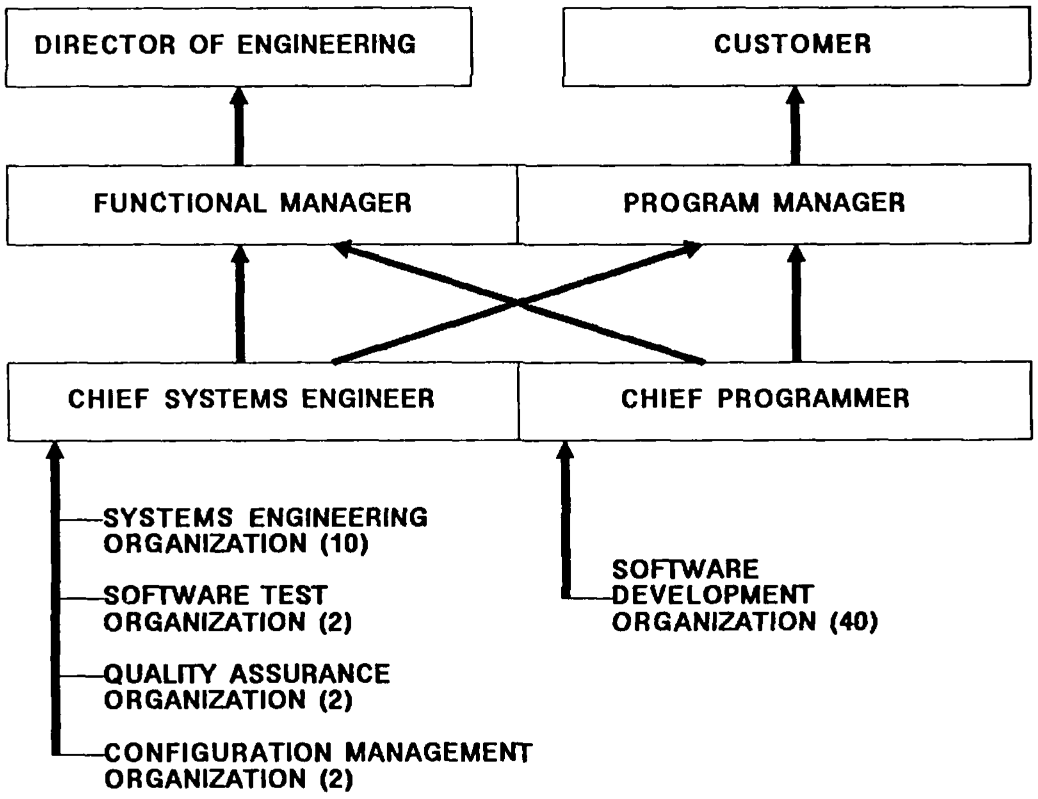

DIRECTOR OF ENGINEERING
CUSTOMER
FUNCTIONAL MANAGER
PROGRAM MANAGER
CHIEF SYSTEMS ENGINEER
CHIEF PROGRAMMER
SYSTEMS ENGINEERING ORGANIZATION (10)
SOFTWARE TEST ORGANIZATION (2)
QUALITY ASSURANCE ORGANIZATION (2)
CONFIGURATION MANAGEMENT ORGANIZATION (2)
SOFTWARE DEVELOPMENT ORGANIZATION (40)

Figure 1: Ada Software Project Organization

The Chief Systems Engineer is responsible for the "system" as an entity. This includes the hardware configuration, user requirements, and all other engineering and support activities that do not fall into the realm of "software development". The Chief Systems Engineer is responsible for the development and the maintenance of the system's requirements, configuration management, quality assurance, and independent system testing.

The Chief Programmer is responsible for all software development activities, including prototyping during the Requirements Analysis phase, the Preliminary Design, the Detailed Design, Coding and Unit Testing, CSC Integration and Testing, and informal CSCI Testing. The Chief Programmer controls the majority of the program's personnel on a software intensive program. It is the Chief Programmer's responsibility to complete the software on time, on schedule, and on budget.

Software Standards and Procedures

Another key aspect of an Ada development that is needed for successful completion is the definition and documentation of the standards and procedures to be used on the project. But, it is not enough to just define and document them. The standards and procedures must be used on a daily basis; they must become the normal way of conducting project business.

A task force has been set up to standardize the use of Ada, the products of each phase, and the tailoring of DoD-STD-2167. The outcome is documented in the DoD-STD-2167 defined Software Standards and Procedures Manual (SSPM). A preliminary SSPM is completed, reviewed, and signed-off by the development team leaders (the Chief Programmer and the CSCI Leaders) early in the development phase. As the software development progresses, the SSPM is enhanced, modified, and elaborated on to form a detailed document that is provided to every employee assigned to the program. This document provides guidance to the software development team in a consistent and detailed manner. The SSPM informs each member of the development team of what, when, and how tools, techniques, standards, procedures, and products are to be used and delivered for each phase.

Requirements Analysis

Software Requirements Analysis accomplishes the functional decomposition of system requirements specified in the System/Segment Specification (SSS) into software requirements that will be implemented in the following Software Development phases.

The organizational structure set up during the Requirements Analysis phase, as shown in Figure 2, allocates the responsibility, for the most part, to the Chief Systems Engineer. Software Requirements Specification Leaders (SRSL) are selected from an experienced group of Software System Engineers (SSE); a group of experienced Software Development Engineers (SDE) are selected to support them.

The responsibility of each SRSL is to generate a software requirements specification for an assigned CSCI along with a leveled set of data flow diagrams, a data dictionary, and a set of testable software requirements in the form of mini-specifications.

The responsibility of the SDE is to assist the SRSL in the development of the SRS and other products, to prototype certain areas of functionality that are not well understood, and to fully understand the requirements so that no time is lost during the Preliminary Design phase.

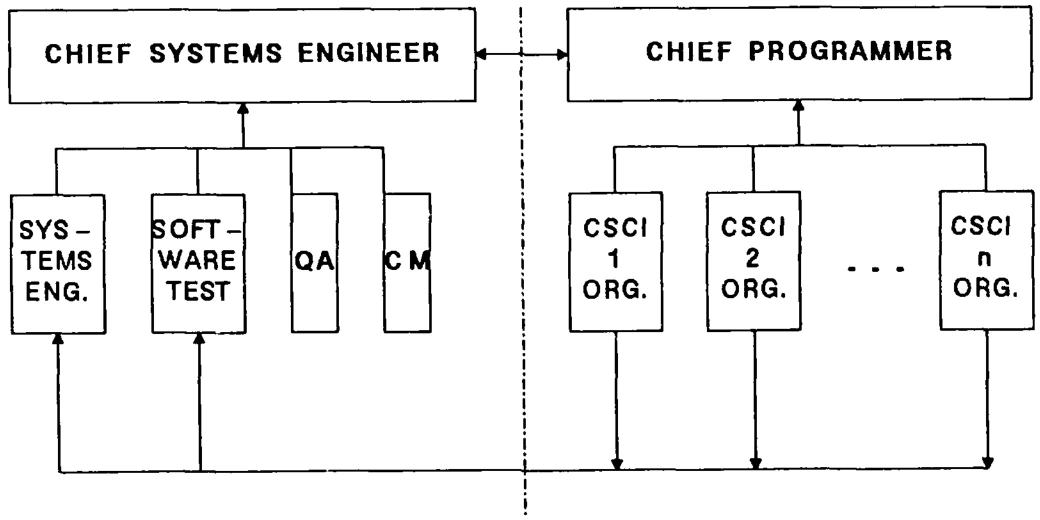


Figure 2: Software Requirements Analysis Organization

As stated previously, Structured Analysis as described by DeMarco was selected as the technique to be used to perform the analysis. The requirements analysis process is described in Figure 3. Most of the SSEs and SDEs already knew the techniques; therefore, it is the most cost-effective technique to use. The only real problem is the transition from Requirements Analysis using structured methods to Preliminary Design using Ada.

The solution to the transition problem is to take a small subset of one CSCI, perform the requirements analysis on it, and perform the transition early in the Requirements Analysis phase. The outcome of this design prototyping activity is a phase transition plan. This plan feeds directly into the Software Standards and Procedures Manual.

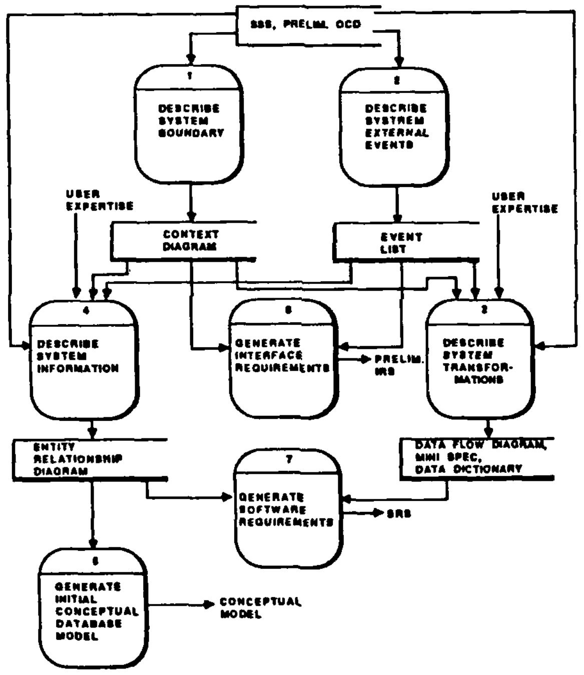

SSS, PRELIM. OCD
1
DESCRIBE SYSTEM BOUNDARY
2
DESCRIBE SYSTREM EXTERNAL EVENTS
USER EXPERTISE
CONTEXT DIAGRAM
EVENT LIST
USER EXPERTISE
4
DESCRIBE SYSTEM INFORMATION
5
GENERATE INTERFACE REQUIREMENTS
PRELIM. IRS
3
DESCRIBE SYSTEM TRANSFOR-MATIONS
ENTITY RELATIONSHIP DIAGRAM
7
GENERATE SOFTWARE REQUIREMENTS
SRS
DATA FLOW DIAGRAM, MINI SPEC, DATA DICTIONARY
6
GENERATE INITIAL CONCEPTUAL DATABASE MODEL
CONCEPTUAL MODEL

Figure 3: Software Requirements Analysis Process

An Ada and VAX Ada environment course has been developed; and it is taught very early during the Requirements Analysis phase. The course is eight weeks long and is taught during lunch hours, during working hours, and after regular working hours. Every employee that is to participate in the program is given the opportunity to attend the course.

A Structured Analysis and Design course has also been developed; and it is taught during lunch hours. Because most of the project personnel have already been exposed to Data Flow Diagrams, Structure Charts,etc., this course was not required by all or needed by all.

Preliminary Design

The purpose of Preliminary Design phase is to develop the top level software architecture and to solidify the interface requirements for each CSCI. Also, any critical elements of a CSCI shall be defined in detail during this phase. The organization set up to perform preliminary design is shown in Figure 4.

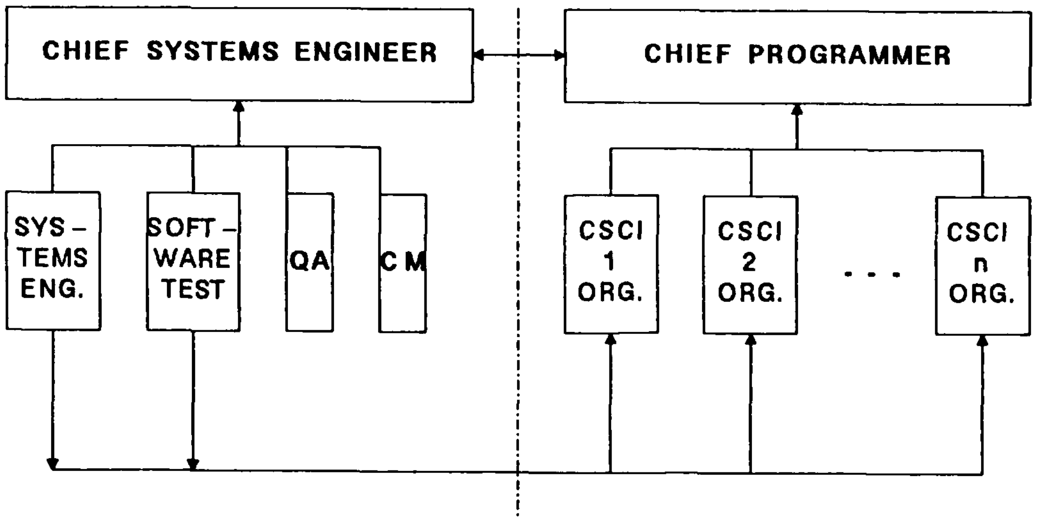


Figure 4: Preliminary Design Organization

During the final preparation of the SRSs and the Software Specification Review (SSR), the SDEs assume the responsibility for the design and development of the CSCIs. Preliminary line of code estimates, schedules, and budgets are generated for each CSCI using the knowledge gained during the previous phase.

As stated above, the main point of the Preliminary

Design phase is to define the top level architecture for a software system, and for each CSCI within it, and to prototype any critical areas to gain a better understanding of how they will be implemented. Figure 5 illustrates the process involved with completing preliminary design. The Software Standards and Procedures Manual (SSPM) defines the work to be completed during the Preliminary Design phase.

The SSPM defines the VAX/VMS directory structure that the CSCI leaders and their subordinates are expected to work within. Each CSCI has an associated VAX account; and the personnel working on that CSCI work in that CSCI's account. The VAX/VMS directory structure definition is shown as Figure 6.

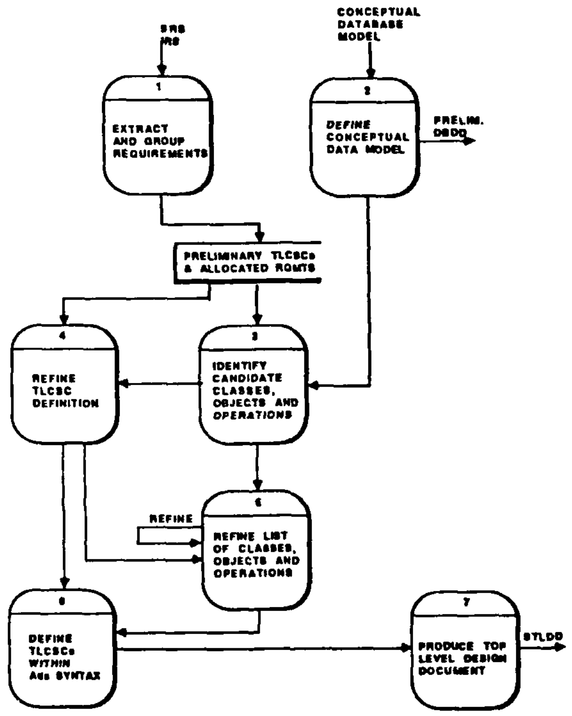

SRS
IRS
CONCEPTUAL DATABASE MODEL
1
EXTRACT AND GROUP REQUIREMENTS
2
DEFINE CONCEPTUAL DATA MODEL
PRELIM. DBDD
PRELIMINARY TLCSCs & ALLOCATED RQMTS
4
REFINE TLCSC DEFINITION
3
IDENTIFY CANDIDATE CLASSES, OBJECTS AND OPERATIONS
6
REFINE
REFINE LIST OF CLASSES, OBJECTS AND OPERATIONS
5
DEFINE TLCSCs WITHIN Ada SYNTAX
7
PRODUCE TOP LEVEL DESIGN DOCUMENT
STLDD

Figure 5: Preliminary Design Process

The Chief Programmer maintains an account with the same directory structure as the CSCI accounts. This account is used to store the final products of the Preliminary Design phase. The Chief Programmer's directory structure is shown as Figure 7. A System level account is also maintained to contain all information common to more than one CSCI. The directory structure for the System account is shown as Figure 8. This account is also controlled by the Chief Programmer.

Most of the SDEs from the Requirements Analysis phase become CSCI Leaders (CSCIL) during the Preliminary Design phase. They now have a detailed understanding of the software requirements that they are to implement; and they have prototyped code and identified common modules or CSCs to use when necessary. Armed with the SSPM, the VAX/VMS directory structure, and the common code, the CSCILs can define the top level software architecture.

A subset of the Ada language is defined for use in defining the processing within a Top-Level CSC (TLCSC) during Preliminary Design; and it is added to the SSPM under Design Standards. Because the processing within a TLCSC is logical in nature, much of the processing will be discarded during the Detailed Design phase. The processing will occur, but it will be performed by Low-Level CSCs (LLCSC) and units. The subset of Ada is used to train the team members on using the VAX/VMS Ada environment.

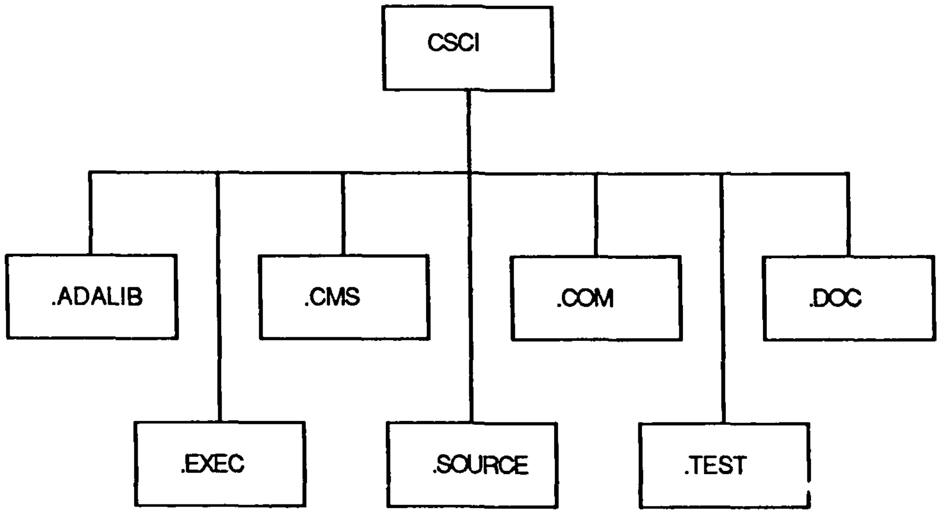


Figure 6: CSCI Accounts Directory Structure

Detailed Design

The purpose of Detailed Design is: (1) to decompose the top level software architecture into Lower level CSCs and units for each Top-Level CSC (TLCSC), and (2) to populate each unit, package, and CSC with inputs, outputs, and process descriptions that satisfy all of the allocated requirements and describe the software in sufficient detail to allow formal coding to begin. The organization set up to perform detailed design is shown in Figure 9. Figure 10 illustrates the detailed design process.

During the final preparation of the Software Top-Level Design Documents (STLDD) during preliminary design, the CSCILs produce new line-of-code estimates and a plan for the Detailed Design phase. This plan includes cost and schedule information as well as a plan for adding additional personnel.

The Detailed Design phase is split into two distinct parts, with a review scheduled between each part.

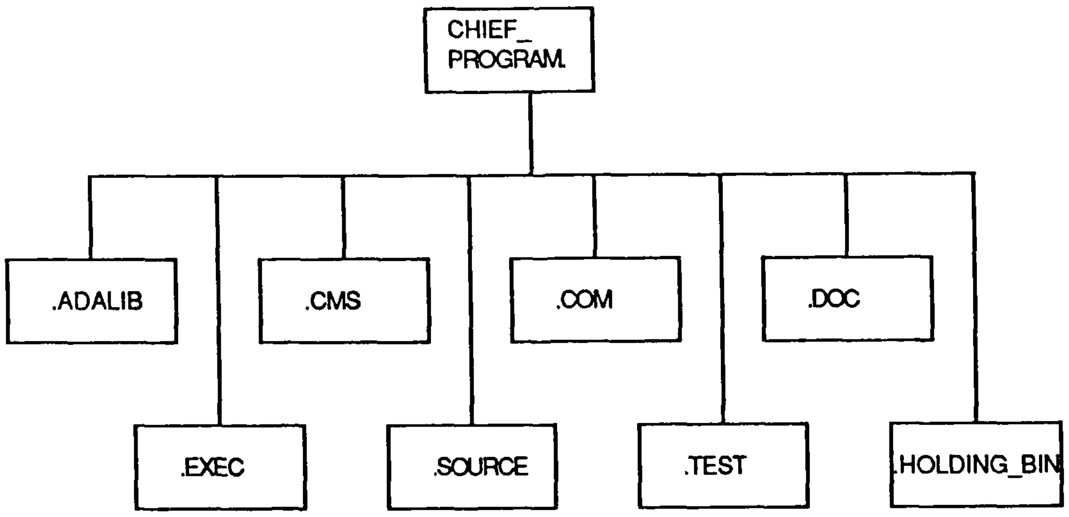


Figure 7: Chief Programmer Account Directory Structure

The first part of detailed design, Design Stage II (preliminary design is considered Design Stage I), completely fills out the DoD-STD-2167 Static Structure; and compiles and links the interfaces across the entire system. This means that Structure Charts are completed for each CSCI, TLCSC, and LLCSC to show the lowest level units as well as all common LLCSCs and units. It also means that the interfaces to every CSCI, TLCSC, LLCSC, and unit are consistent and correct. The Design Stage II review is used to assess the quality of the completed software architecture and to approve the formal start of Design Stage III.

Design Stage III accomplishes the design of algorithms and processing for each of the LLCSCs and units. Because each of the interfaces have already been checked for consistency (via compiling and linking), each unit can now be treated independently and designed in detail. At the end of Design Stage III, The DoD-STD-2167 Static Structure is completed, compiled, and linked and the detailed design is ready for the Critical Design Review (CDR).

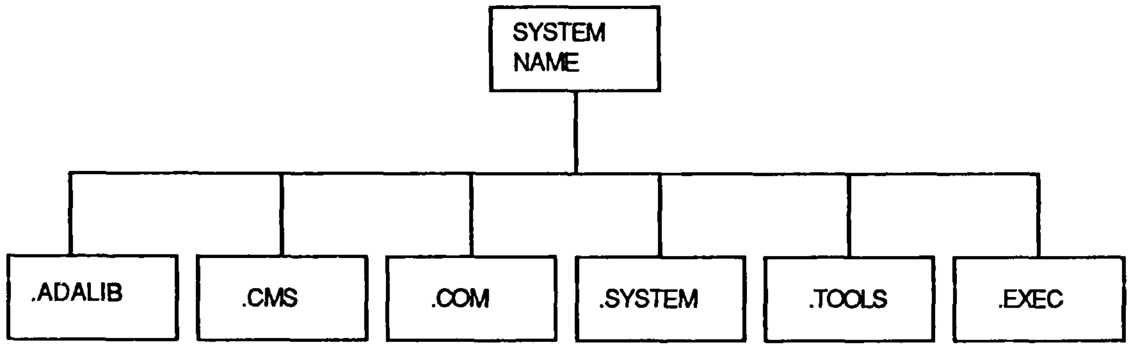


Figure 8: System Account Directory Structure

Code and Unit Testing

The purpose of Coding and Unit Testing is the creation of the software products. Coding is the expansion of the Ada constructs and code produced during the Detailed Design phase into its final detailed compiled and tested form. Unit Testing is the lowest level of testing performed during the software development phase. The organization set up to perform coding and unit testing is show on Figure 9 in the previous paragraph. The organization is stable during detailed design and coding and unit testing because they deal with the same entities, at a similar level of detail.

During the final preparation of the Software Detailed Design Documents, and any other products of the Detailed Design phase, the CSCILs again produce new line-of-code estimates and a plan for the implementation of the Code and Unit Testing phase. This plan includes the same information as provided previously, at the start of the Detailed Design phase, but more current. Figure 11 illustrates the process involved with the Code and Unit Testing phase.

The CSCILs assign coding tasks to new personnel by assigning them to a CSC Leader (CSCL). A CSCL is responsible for the technical integrity of one or more CSCs of a CSCI or common to more than one CSCI.

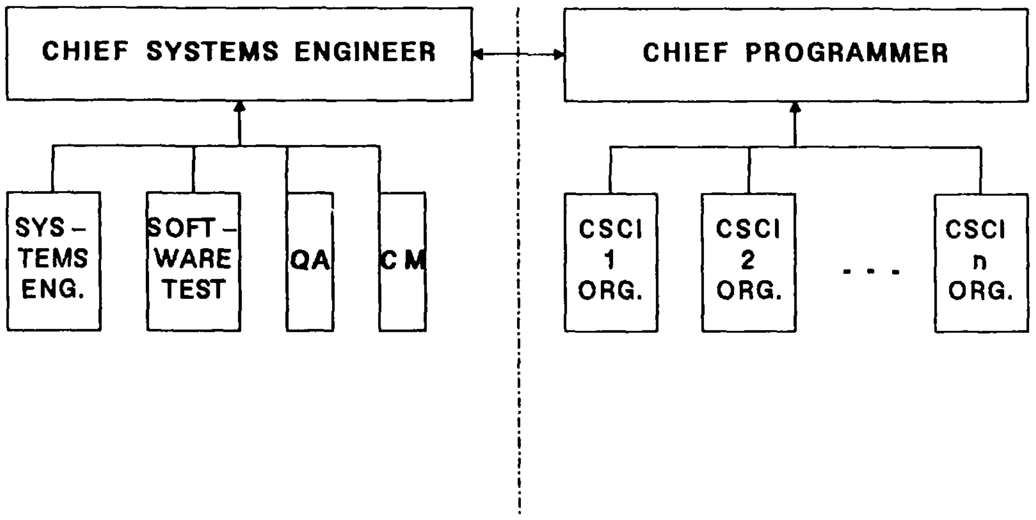


Figure 9: Detailed Design and Code and Unit Test Organization

Each Ada coder has a schedule that lays out coding and unit testing responsibilities. Each Ada coder must "Check Out" a unit from the Software Development Library (SDL), which is the physical DoD-STD-2167 Static Structure that is stored on disk and formally controlled by Configuration Management (CM). Once a coder has completed the coding and compiling of a unit, a unit test procedure must be completed according to the definition and format described in the SSPM. When the unit test procedure is completed, the coder schedules a code walkthrough with the CSCL, the CSCIL, Quality Assurance, and, at times, the Chief Programmer. Following the successful code walkthrough, the Ada coder performs the unit test and then, "Checks In" the unit, the unit test procedure, and the unit test results to the SDL. The Ada coder is now free to proceed to the next assigned unit on the schedule.

Computer Software Component (CSC) Integration and Testing

The purpose of CSC Integration and Testing is to integrate units into CSCs and to test their interaction against a documented test procedure. Figure 12 shows the organization as it applies to CSC Integration and Testing. Figure 13 shows the process used to complete the CSC Integration and Testing.

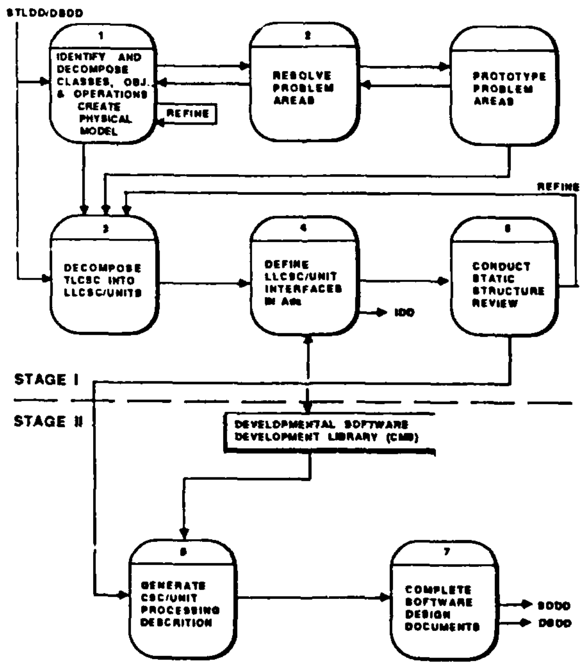

STLDD/DSDD
1
IDENTIFY AND DECOMPOSE CLASSES, OBJ. & OPERATIONS CREATE PHYSICAL MODEL
REFINE
2
RESOLVE PROBLEM AREAS
PROTOTYPE PROBLEM AREAS
REFINE
3
DECOMPOSE TLCSC INTO LLCSC/UNITS
4
DEFINE LLCSC/UNIT INTERFACES IN Ada
IDD
5
CONDUCT STATIC STRUCTURE REVIEW
STAGE I
STAGE II
DEVELOPMENTAL SOFTWARE DEVELOPMENT LIBRARY (CMB)
6
GENERATE CSC/UNIT PROCESSING DESCRITION
7
COMPLETE SOFTWARE DESIGN DOCUMENTS
SDDD
DSDD

Figure 10: Detailed Design Process

Once the coding and unit testing has been completed, the units must be integrated into CSCs. A CSC is a collection of units that, together, perform some subset of the total CSCI requirements. Therefore, CSC integration should concentrate on completing integrated functions that perform a subset of the total work of the CSCI. The testing of an integrated CSC uses tests from unit testing and from CSCI testing as well as tests that are defined specifically to test functionality that might normally not be visible to the test engineer.

A CSCL is normally responsible for the integration and testing of a CSC. The CSCL must "Check Out" all of the units to be integrated from the SDL. The CSCL generates a CSC Test

Procedure, and defines and generates any software needed to create the correct test environment. Upon completion of the CSC test environment, the CSCL schedules a walkthrough with the CSCIL, QA, and the Chief Programmer for large CSCs. Once the CSC has been integrated and completely tested, the CSC, the test procedures, the test environment, and the test results are "Checked Into" the SDL.

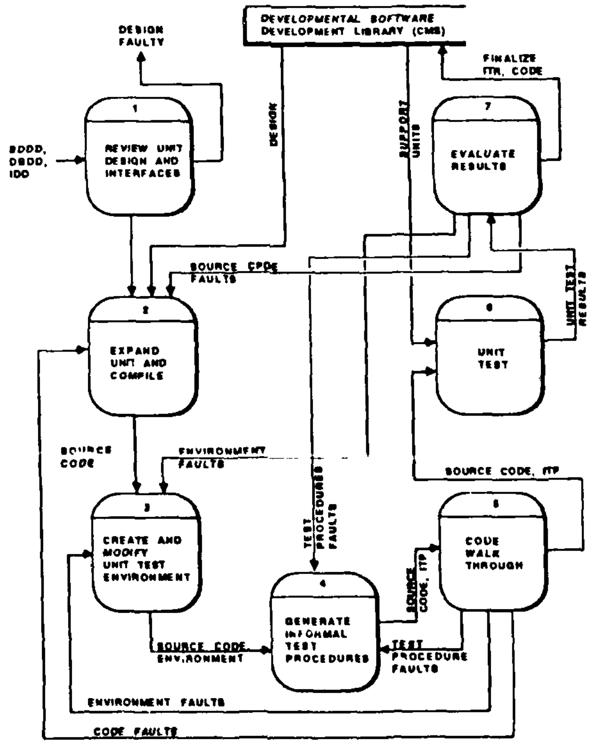


Figure 11: Code and Unit Testing Process

After a CSCL has completed the integration and testing of all scheduled CSCs, the CSCL is moved to support the Software Test Organization. This move guarantees that the Software Test Organization will not be "stuck with" the Software Development Organization's problems. It is in the best interest of the CSCL to test CSCs thoroughly and completely. Figure 12 illustrates the transition from CSCL to Software Test Engineer (STE).

Computer Software Configuration Item (CSCI) Testing

The purpose of CSCI Testing is to formally qualify each CSCI against its associated Software Requirements Specification (SRS) and the Interface Requirements Specification (IRS), using the formal Test Plan and Test Procedures written specifically for the CSCI. Figure 14 illustrates the CSCI Testing organization.

The Software Test organization is made up of Independent Software Test Specialists (ISTS) and Software Development Engineers (SDE). The ISTSs control the CSCI testing process, and ensure that unbiased testing is performed. The SDEs have the indepth knowledge of the system that is required for detailed testing and for troubleshooting when tests fail. Figure 15 shows the process followed for performing CSCI testing.

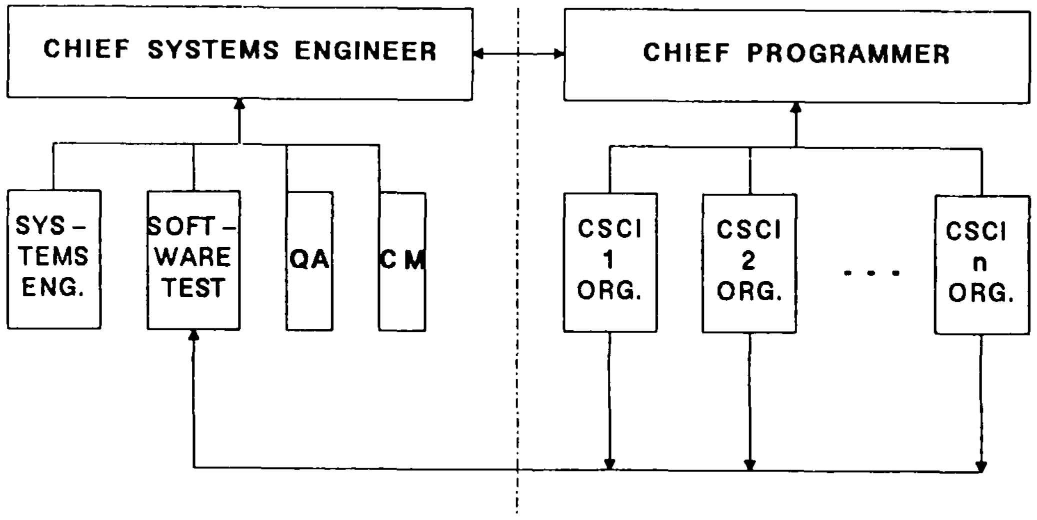


Figure 12: CSC Integration and Test Organization

The responsibility for CSCI testing rests with the Software Test Leader (STL), and ultimately, with the Chief Systems Engineer (CSE). At this point in the development, the Chief Programmer has successfully completed the software development and integration, and has formally "sold off" to the Chief Systems Engineer. If the CSE's organization has been successful, the software will be under formal control in the VAX/VMS directories defined in the SSPM, QA will have formally accepted all software and documentation products, and the Software Test organization will have government accepted test procedures with which each CSCI will be tested. The responsibility now rests with the CSE (and the Program Manager) to formally "sell off" to the customer. Assuming that the CSE's organization has been successful, the Ada software program will conclude with successful formal testing and "sign off" by the customer.

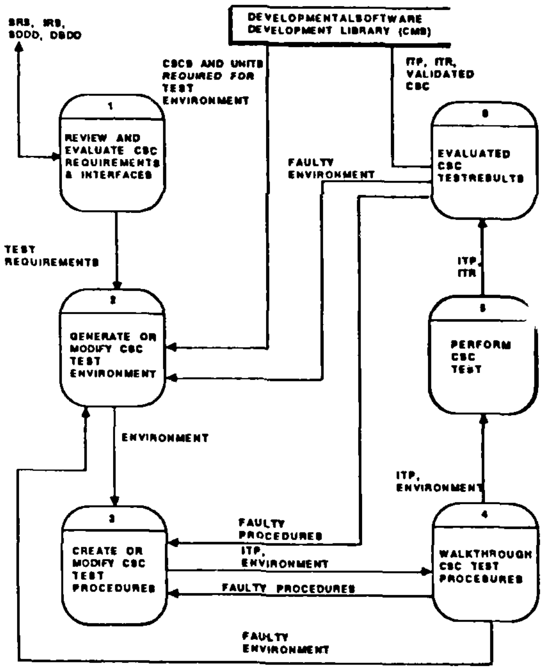

SRS, IRS, SDDD, DSDD
DEVELOPMENTALSOFTWARE DEVELOPMENT LIBRARY (CMS)
CSCS AND UNITS REQUIRED FOR TEST ENVIRONMENT
ITP, ITR, VALIDATED CSC
1
REVIEW AND EVALUATE CSC REQUIREMENTS & INTERFACES
6
EVALUATED CSC TESTRESULTS
FAULTY ENVIRONMENT
TEST REQUIREMENTS
ITP, ITR
2
GENERATE OR MODIFY CSC TEST ENVIRONMENT
5
PERFORM CSC TEST
ENVIRONMENT
ITP, ENVIRONMENT
3
CREATE OR MODIFY CSC TEST PROCEDURES
FAULTY PROCEDURES
ITP, ENVIRONMENT
FAULTY PROCEDURES
4
WALKTHROUGH CSC TEST PROCEDURES
FAULTY ENVIRONMENT

Figure 13: CSC Integration and Test Process

CONCLUSION

This paper describes a methodology that has been proven on a large Ada program. But, success does not and should not mean that the methodology is unchanging. In fact, change is part of the methodology. The authors of this paper believe that the methodology defined here is a baseline from which change can occur in an intelligent and planned manner.

The thrust of the methodology, as stated previously, is to quickly get personnel on-board, to build a projectized software organization, to teach the personnel the tools, techniques, and procedures and the Ada language, and to define and standardize the use of Ada, the products of each phase, and the tailoring of DoD-STD-2167. These steps will be accomplished for any Ada program. But the contents of some of the steps may change at the beginning of a new project.

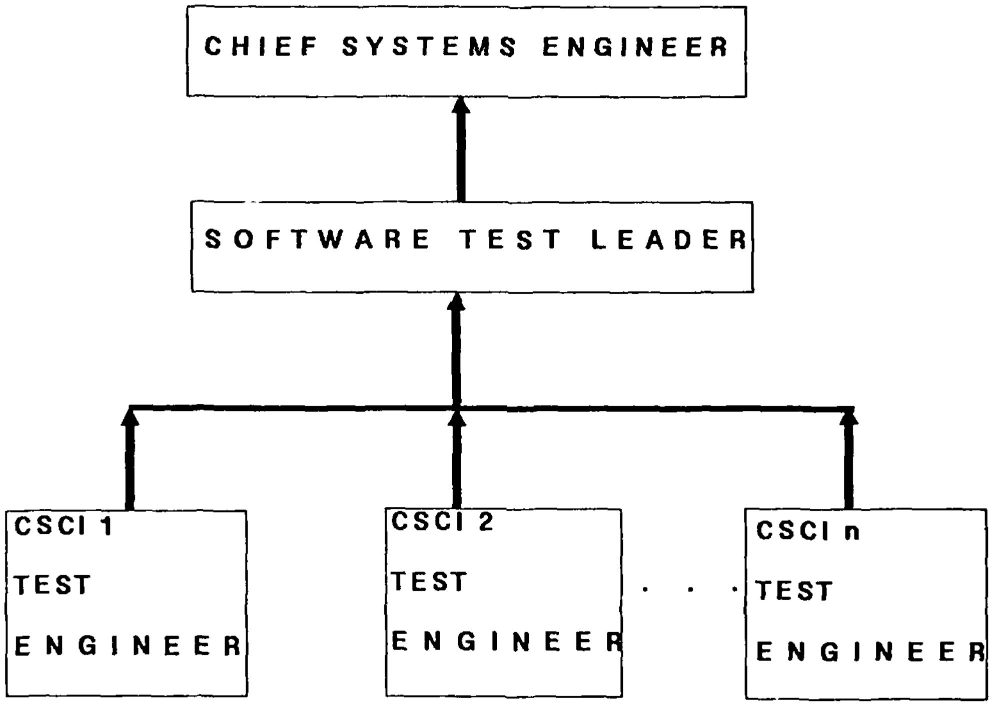


Figure 14: CSCI Testing Organization

The software development organization will remain the same across any project. The one defined in Figure 1 provides both tight control and creative freedom (within boundaries). Each individual on the organization understands their role in and responsibility to the success of the project.

The tools, techniques, and procedures may change for a new project if the organization is mature and understands the importance of educating the entire organization in the new ones prior to the start of software development. Once a well defined method for performing object-oriented requirements analysis is documented, it can be incorporated into the methodology. But, there are risks. The organization must understand that the phases of the project are linked together and cannot be treated separately. If the techniques used to perform requirements analysis change, so must the techniques used to perform preliminary design, detailed design, and so on. But all this means is that the use of new tools, techniques, and procedures must be completely defined and documented (in the SSPM); and the organization must be trained in the use of them.

The standards will change because their definition is evolutionary in nature. Knowledge and insight will be gained with the completion of each phase of each project. Therefore, a "lessons-learned" document should be compiled at the end of each phase. This document should be archived and incorporated into the SSPM. If done properly, an audit trail will exist from the baseline methodology to the present one. This will allow an organization to recover from a methodology decision that adversely affects the ability to complete a project correctly, on time, on schedule, and on budget.

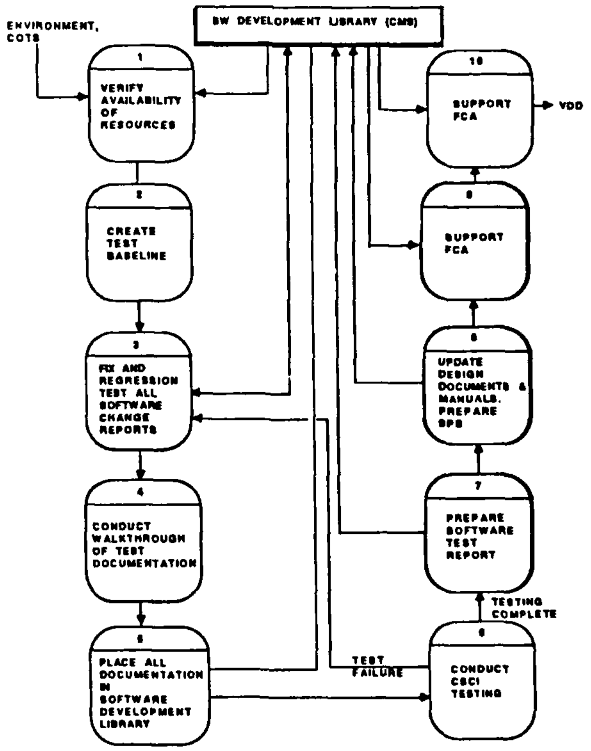

SW DEVELOPMENT LIBRARY (CMS)
ENVIRONMENT, COTS
1
VERIFY AVAILABILITY OF RESOURCES
2
CREATE TEST BASELINE
3
FIX AND REGRESSION TEST ALL SOFTWARE CHANGE REPORTS
4
CONDUCT WALKTHROUGH OF TEST DOCUMENTATION
5
PLACE ALL DOCUMENTATION IN SOFTWARE DEVELOPMENT LIBRARY
6
CONDUCT CSCI TESTING
TEST FAILURE
TESTING COMPLETE
7
PREPARE SOFTWARE TEST REPORT
8
UPDATE DESIGN DOCUMENTS & MANUALS. PREPARE SPS
9
SUPPORT FCA
10
SUPPORT FCA
VDD

Figure 15: CSCI Testing Process

In conclusion, the methodology presented here is not a complicated one; and, it does not present a new or innovative technique. It does define the basic needs that must be satisfied to ensure that an Ada project can be completed on time, etc. Those needs can be synopsized as follows:

1. get personnel on-board early

2. assign key personnel and define the responsibilities of each completely

3. create and teach early and often the tools, techniques, and procedures to be used on the program

4. define as early as possible the standards to used for the project for Ada, the products of each phase, and DoD-STD-2167; and document them in an SSPM.

If a software organization can focus on these issues early, an Ada project, whether its the first or the twenty-first, has a very good chance of completing successfully.

REFERENCES

1. Boehm, B.; Software Engineering Economics; 1981

2. Booch, G.; Software Engineering with Ada; 1983.

3. DeMarco, T.; Structured Analysis and System Specification; 1979.

4. Department ofDefense; Military Standard for Defense System Software Development, DoD-STD-2167(A); 1988.

5. MIL-STD-1815A; Ada Programming Language Reference Manual.

6. Ward, P. and Mellor, S.; Structured Development for Real-Time Systems; 1985.

TRAINING PROGRAMMERS IN ADA USING A GRADUATED EXAMPLE

KARL REHMER, LINDA RISING, JAMES SILVER
MAGNAVOX ELECTRONIC SYSTEMS COMPANY
1313 PRODUCTION ROAD
FORT WAYNE, IN 46808

ABSTRACT

This paper describes the authors' experiences using a single example throughout a series of Ada training sessions. The example is enhanced as features of the language are introduced.

INTRODUCTION

In 1986 and 1987, the authors were involved in the development of Ada training sessions at Magnavox Electronic Systems Company in Fort Wayne, Indiana. The course was divided into four parts covering: (1) the Pascal-like subset - 4 hours; (2) user-defined types and beginning data structures - 8 hours; (3) packages and generics - 4 hours and (4) exceptions and tasking - 8 hours. The classes met for two-hour periods twice a week.

The students were programmers whose prior programming work had been primarily in assembly language. Some had never programmed in a high-level language, while others had experience with FORTRAN and Pascal. Since these programmers had other responsibilities, it was essential that the sessions be as compact and efficient as possible. The students also had a limited amount of time available to work on exercises outside of class. Since the students were about to begin work on a project which required Ada, the course needed to give them a sound foundation, not only in the programming language, but in principles of software engineering, allowing them to use the language to its fullest.

The course needed to address two apparently conflicting needs. First, it had to make the most efficient possible use of time in the classroom. This greatly restricted the amount of time which could be spent developing examples. Second, since the students were to begin designing and programming in Ada, the inclusion of significant portions of exemplary code was essential. The authors adopted the strategy of using a single example throughout the course. This eliminated the time required to create a setting for many examples.

The example needed to be sufficient to illustrate all of the syntactic features of the language as they were developed, but it also had to be useful for illustrating appropriate software engineering principles and more advanced language features such as generics and tasking.

THE EXAMPLE

A complex calculator evaluates complex-valued expressions. Complex numbers are entered as a sequence of two integers, enclosed by parentheses, immediately preceded by the character E, for "ENTER". The first integer represents the real part, and the second integer the imaginary part of the complex number. The restriction to integer was done to defer consideration of the various floating point types in Ada. Expressions are entered in postfix notation; only the operations addition, subtraction and multiplication will be allowed. Each expression is to be entered on a single line and terminated by '='. Input is assumed correct.

```
SAMPLE INPUT                          SAMPLE OUTPUT
E(3 4)E(2 6)+E(2 5)*=                 (-40 45)
```

DESIGN

The training sessions emphasized the object-oriented approach proposed by Grady Booch [2]. The objects to be considered in this example are: complex number and stack. They illustrate the simplest application for Ada packages, that of representing an abstract data type, the logical grouping of a type and its associated operations.

Operations identified for complex number are:

Read, Write, Add, Subtract, Multiply,
Create, Extract_Real_Part, Extract_Imaginary_Part

The latter three operations are necessary to avoid using implementation details outside the package.

Operations identified for stack are:

Pop, Push, Create, Clear, Empty, Full

Two packages are used to hide the implementation details of each data structure and its associated operations.

The Ada langauge does not implement I/O for the primitive types in the same package (Standard) where these types are defined. Instead, Text_IO or an appropriate instantiation of a generic package in Text_IO is used. Thus, it was decided to separate the operations involved in complex I/O from the rest of the operations on complex numbers. A separate Complex_IO Package could be written but in later examples when the packages are enhanced to include the use of generics, the "use hierarchy" becomes too complex. To encourage reusability, the most general kinds of packages should be created. The I/O format used in this example might not be used in any other situation. Thus, Read_Complex and Write_Complex are included in the main driver. If the I/O operations were of a general nature, a good argument could be made for including them in the Complex Number Package.

FIRST IMPLEMENTATION

It has been said [3] that packages are what Ada is all about. The first implementation attempts to present a program which takes advantage of the resources in two packages. Most students have a notion of a stack and of complex number and the concept of developing these separate entities and storing them in a library to be used by any appropriate driver has immediate appeal. The first simple calculator is intuitively clear and presents constructs, procedures and functions encountered in Part I, the Pascal-like subset. Only primitive data types are used. The emphasis is on the use of the simplest Ada components in a complete program setting. The program in Listing 1 in the Appendix was the first solution to the problem.

SECOND IMPLEMENTATION

While the idea of a package and development for reusability are appealing concepts for working programmers, realizing that any user can manipulate carefully designed details, violates the theory of abstraction. Students are lead to see the problems in the simple example in this regard and are introduced to Ada's private and limited private types.

To prevent outside access of any implementation details of a type defined inside a package, the type may be declared as 'private'. The only operations available on private types are those provided in the package, assignment and equality (inequality by default).

An implementor can restrict access to a type even further by declaring a type 'limited private'. This makes assignment and equality (and therefore inequality) forbidden operations. In Listing 2, the second version of the program declares a complex number as a private type and makes the stack a limited private type. If the stack were implemented as a linked list, assignment would not create a new stack but would create a new pointer to the same stack and equality would not check for equality of the stack but only for equality in the values of the top pointer. Declaring the stack to be limited private eliminates these meaningless operations.

THIRD IMPLEMENTATION

Reusability is a major topic in proper software engineering. The need to avoid "re-invention of the wheel" is a major concern of working programmers. This implementation is used to show the students how generic packages can be used to create reusable modules. The stack package was previously written specifically for complex numbers, however, the algorithms used are the same as those needed for any stack. In most programming languages, separate collections of stack operations would have to be written, one for each data type. Having collections of modules perform the same operations is wasteful in the use of space and programmer time. It is also error-prone, since corrections in one set of operations might not be made in all sets. Generic units in Ada are templates from which several specific instances can be created without duplication of effort.

In addition, complex numbers would probably not be composed of two integer values. Some real type would be a more likely choice, thus, the example generic package for complex numbers allows instantiation of a new complex package for a floating-point type. The generic solution is found as Listing 3.

FOURTH IMPLEMENTATION

In order to effectively program in Ada, the students need to be familiar with exceptions and exception handlers which allow programs to cope with run-time errors. Exceptions may be raised explicitly with a raise statement, or raised automatically during program execution.

The fourth implementation, in Listing 4, shows the students a number of different features and uses of exception handlers. The complex number package exports an exception named Complex_Numeric _Error. The body of the package illustrates several simple error handlers. A Numeric_Error raised by the run-time support system is 'handled' simply by raising a Complex_Numeric_Error. This allows the caller to distinguish in its error handling between an error which occurred inside the complex number package and one which occurred during some other numeric operation.

The Stack package exports two exceptions. One indicates that an overflow occurred during a Push. The second is raised on an attempt to Pop an element from an empty stack.

The new version of the calculator procedure defines three additional exceptions of its own. These are raised internally and pass information to the exception handler, allowing it to print an appropriate diagnostic for the user. It also illustrates a typical approach to error handling. The loop which evaluates a single expression is enclosed in Evaluation_Block. This block contains an error handler which copes with any error occurring during the evaluation of one expression by printing a diagnostic message for the user, clearing the stack to begin evaluation of the next expression, and discarding the rest of the line which caused the error. The main procedure includes an error handler for an End_Error which allows the user to terminate execution at any time by signalling end of file.

FIFTH IMPLEMENTATION

The students also needed exposure to the Ada tasking features for the development of concurrent programs. These may be used to exploit parallel architectures, multiprocessor configurations, or facilitate implementation of designs in which concurrency has been used to provide a conceptually simpler solution.

The fifth implementation, found in Listing 5, uses a task to allow evaluation of operators in an expression to proceed concurrently with reading later parts of that expression. Since the result of a particular operation is always pushed onto the stack, often not to be used until much later, it seems reasonable to design a solution which continues to read and process the expression without waiting for the completion of the operation. This design can be implemented by providing a task to simulate the operation of an arithmetic unit. It provides the following entries:

    Clear_Unit, Add, Subtract, Multiply, Get_Result

The task copies the values of the operands to local variables and ends the rendezvous. This allows the calling task to become ready while the Arithmetic Unit continues its evaluation. Since the results of earlier operations will be needed in Last-In-First-Out order, the task maintains these values in its own stack.

The main program evaluates expressions in the following way: Numbers read in are simply pushed onto a stack, if an operator is read, the two operands are passed to the appropriate entry of the Arithmetic Unit task. Since the result will not be retrieved until actually needed, the main program pushes a token on the stack indicating that the entry is pending. When such a token is popped off this stack, the number is retrieved from the arithmetic unit by calling Get_Result. The Clear_Unit entry is provided to allow the exception handler to reinitialize the arithmetic unit when an error occurs.

This implementation serves to illustrate the use of tasks in Ada programs. However, it should be noted that, in a typical implementation of Ada running on a single processor, the Arithmetic Unit will keep control of the processor until it has completed the evaluation of the operation and pushed the result on its own stack. The concurrency is conceptual, not actual. Nevertheless, this same design would result in substantial concurreny if the operand type were a large array and a multiprocessor system were available.

SIXTH IMPLEMENTATION

Ada allows tasks to be treated as objects. When defined as a task type, many identical copies of a task can be created; these can be combined with other objects in a record structure, and, when used with access types, can be created dynamically. The last implementation exploits this feature by defining three different task types and using these in a variant record for the stack entries. It defines an access type to this record type so that the tasks can be created dynamically as needed. Since the tasks are part of a record structure and can be treated as objects, the stack is conceptually simpler than the last implementation. Each stack entry is either a 'register' which contains a constant or a task computing the result of an operation. The Pop procedure simply identifies the item on top of the stack and returns the appropriate value. This final implementation is found in Listing 6.

CONCLUSION

Magnavox has been involved with Ada training in the past but it has been more concentrated, typically involving week-long sessions of eight-hour classes/lab. Since two of the authors had participated in these Ada training sessions as students and since the authors are all full-time professors with teaching experience, it was considered a decided advantage to be able to offer Magnavox the more flexible, shorter class periods.

The advantages to this more flexible scheduling are many: (1) as educators with experience teaching classes of two-hours or less, the authors agree that students have the greatest chance of learning if the material is presented in sessions of no more than two hours; (2) the students in our classes were programmers and managers with outside responsibilities, who could afford to spend a couple of hours without much difficulty but would find eight-hour sessions impossible; this has forced many of them in the past to miss substantial portions of classes rendering subsequent sessions almost impossible to comprehend; (3) the intervening periods of time gave students time to think about the ideas presented in previous classes and a chance to work on some exercises suggested in class, so they were more able to assimilate the next two-hour presentation and absorb as much as possible from it.

Having the underlying complex calculator example throughout the series of classes gave the students some point of reference. Introducing new concepts in a familiar setting was less confusing to the students and provided them with complete examples of Ada programs using each new feature, something most text books are unable to provide. These examples could be typed and tested and modified by the students outside the class. Many of the students did this and their results and questions provided a good beginning for each class session. The use of this example to introduce Ada concepts was a successful teaching device.

REFERENCES

1. Booch, G., Software Engineering with Ada, Benjamin Cummings, 1983.

2. Booch, G., "Object-Oriented Development", IEEE Trans. Soft. Eng., vol SE-12, No. 2, pp. 211-222, Feb. 1986.

3. Barnes, J.G.P., Programming in Ada, Second Edition, Addison-Wesley Publishing Company, London, 1984.

APPENDIX

Listing 1

```
package Complex_Number_Package is
  type Complex_Number is
    record
      Real_Part : Integer;
      Imag_Part : Integer;
    end record;

  function Create( Real_Part : in Integer;
                   Imag_Part : in Integer )
    return Complex_Number;

  function Real_Part( Number : in Complex_Number )
    return Integer;

  function Imag_Part( Number : in Complex_Number )
    return Integer;

  function "+"( First_Number  : in Complex_Number;
                Second_Number : in Complex_Number )
    return Complex_Number;

  function "-"( First_Number  : in Complex_Number;
                Second_Number : in Complex_Number )
    return Complex_Number;

  function "*"( First_Number  : in Complex_Number;
                Second_Number : in Complex_Number )
    return Complex_Number;
end Complex_Number_Package;

package body Complex_Number_Package is

  function Create( Real_Part : in Integer; Imag_Part : in Integer )
    return Complex_Number is

  New_Number : Complex_Number;

  begin  -- Create
    New_Number.Real_Part := Real_Part;
    New_Number.Imag_Part := Imag_Part;
    return New_Number;
  end Create;

  function Real_Part( Number : in Complex_Number )
    return Integer is

  begin  -- Real_Part
    return Number.Real_Part;
  end Real_Part;
```

```
   function Imag_Part( Number : in Complex_Number )
     return Integer is

   begin  -- Imag_Part
     return Number.Imag_Part;
   end Imag_Part;

   function "+"( First_Number  : in Complex_Number;
                 Second_Number : in Complex_Number )
     return Complex_Number is

     New_Number : Complex_Number;

   begin  -- "+"
     New_Number.Real_Part := First_Number.Real_Part +
                             Second_Number.Real_Part;
     New_Number.Imag_Part := First_Number.Imag_Part +
                             Second_Number.Imag_Part;
     return New_Number;
   end "+";

   -- code for "-" and "*" is similar
 end Complex_Number_Package;

   with Complex_Number_Package;
 package Stack_Package is
   Max_Size  : constant Integer := 100;
   subtype Top_Type is Integer range 0..Max_Size;
   subtype Stack_Index_Type is Integer range 1..Max_Size;
   type Stack_Array is array( Stack_Index_Type ) of
     Complex_Number_Package.Complex_Number;
   type Stack_Type is
     record
       Top  : Top_Type := 0;
       Info : Stack_Array;
     end record;

   procedure Clear( Stack : in out Stack_Type );

   function Empty( Stack : in Stack_Type )
     return Boolean;

   function Full( Stack : in Stack_Type )
     return Boolean;

   procedure Pop( Element :    out Complex_Number_Package.Complex_Number;
                  Stack   : in out Stack_Type );

   procedure Push ( Element : in     Complex_Number_Package.Complex_Number;

 end Stack_Package;
```

```
package body Stack_Package is

  procedure Clear( Stack : in out Stack_Type ) is

  begin  -- Clear
    Stack.Top := 0;
  end Clear;

  function Empty( Stack : in Stack_Type )
    return Boolean is

  begin  -- Empty
    return Stack.Top = 0;
  end Empty;

  function Full( Stack : in Stack_Type )
    return Boolean is

  begin  -- Full
    return Stack.Top = Max_Size;
  end Full;

  procedure Pop( Element :    out Complex_Number_Package.Complex_Number;
                 Stack   : in out Stack_Type ) is

  begin  -- Pop
    Element := Stack.Info( Stack.Top );
    Stack.Top := Stack.Top - 1;
  end Pop;

  -- code for Push is similar to Pop
end Stack_Package;

  with Text_IO;
  with Stack_Package;
  with Complex_Number_Package;
  use Complex_Number_Package;    -- to allow infix notation
procedure Calculator is
  Stack          : Stack_Package.Stack_Type;
  Next_Character : Character;
  Complex_Value, Left_Operand, Right_Operand, Result
                 : Complex_Number_Package.Complex_Number;
  package Int_IO is new Text_IO.Integer_IO( Integer );

  procedure Read_Complex (
    Number : out Complex_Number_Package.Complex_Number ) is

    Real_Part, Imag_Part : Integer;
    Delimiter : Character;
  begin  -- Read_Complex
    Text_IO.Get( Delimiter );   Int_IO.Get( Real_Part );
    Int_IO.Get( Imag_Part );    Text_IO.Get( Delimiter );
    Number := Complex_Number_Package.Create( Real_Part, Imag_Part );
  end Read_Complex;
  procedure Write_Complex (
    Number : in Complex_Number_Package.Complex_Number ) is

    Real_Part, Imag_Part : Integer;

  begin  -- Write_Complex
    Real_Part := Complex_Number_Package.Real_Part( Number );
    Imag_Part := Complex_Number_Package.Imag_Part( Number );
    Text_IO.Put( '(' );  Int_IO.Put( Real_Part, 1 );
    Text_IO.Put( ' ' );  Int_IO.Put( Imag_Part, 1 );
    Text_IO.Put( ')' );  Text_IO.New_Line;
  end Write_Complex;

begin  -- Calculator
  Evaluate_All_Expressions:
  while not Text_IO.End_Of_File loop
    Text_IO.Get( Next_Character );
    Evaluate_One_Expression:
    while Next_Character /= '=' loop
      case Next_Character is
        when 'E' =>
          Read_Complex( Complex_Value );
          Stack_Package.Push( Complex_Value, Stack );
        when '+' | '-' | '*' =>
          Stack_Package.Pop( Right_Operand, Stack );
          Stack_Package.Pop( Left_Operand, Stack );
          case Next_Character is
            when '+' =>
              Result := Left_Operand + Right_Operand;
            when '-' =>
              Result := Left_Operand - Right_Operand;
            when '*' =>
              Result := Left_Operand * Right_Operand;
            when others =>
              null;
          end case;
          Stack_Package.Push( Result, Stack );
        when others =>
          null;
      end case;
      Text_IO.Get( Next_Character );
    end loop Evaluate_One_Expression;
    Stack_Package.Pop( Result, Stack );
    Write_Complex( Result );
    Stack_Package.Clear( Stack );
  end loop Evaluate_All_Expressions;
end Calculator;
```

Listing 2

```
package Complex_Number_Package is
  type Complex_Number is private;

  -- specifications for functions as above
```

```
    private
      type Complex_Number is
        record
          Real_Part   : Integer;
          Imag_Part   : Integer;
        end record;
  end Complex_Number_Package;

   with Complex_Number_Package;
package Stack_Package is
  type Stack_Type is limited private;

  -- specifications for functions and procedures as above

  private
    Max_Size   : constant Integer := 100;
    subtype Top_Type is Integer range 0..Max_Size;
    subtype Stack_Index_Type is Integer range 1..Max_Size;
    type Stack_Array is array( Stack_Index_Type ) of
      Complex_Number_Package.Complex_Number;
    type Stack_Type is
      record
        Top   : Top_Type := 0;
        Info  : Stack_Array;
      end record;
end Stack_Package;
```

Listing 3

```
generic
  type Float_Type is digits<>;
package Complex_Number_Generic_Package is
  type Complex_Number is private;

  -- specification for functions similar to above
  -- with Float_Type instead of Integer

  private
    type Complex_Number is
      record
        Real_Part : Float_Type;
        Imag_Part : Float_Type;
      end record;
end Complex_Number_Generic_Package;

package body Complex_Number_Generic_Package is
  -- code as above but with Float_Type instead of Integer

end Complex_Number_Generic_Package;

generic
  type Item is private;
package Stack_Package is
  type Stack_Type is limited private;
  -- specification as above with Item instead of Complex_Number
  private
    Max_Size  : constant Integer := 100;
    subtype Top_Type is Integer range 0..Max_Size;
    subtype Stack_Index_Type is Integer range 1..Max_Size;
    type Stack_Array is array( Stack_Index_Type ) of Item;
    type Stack_Type is
      record
        Top  : Top_Type := 0;
        Info : Stack_Array;
      end record;
end Stack_Package;

package body Stack_Package is
  -- code as above with Item instead of Complex_Number
end Stack_Package;

  with Text_IO;
  with Stack_Package;
  with Complex_Number_Package;
procedure Calculator is
  package Complex_Package is new Complex_Number_Package( Float );
  use Complex_Package;
  package Complex_Stack is new Stack_Package(
    Complex_Package.Complex_Number );
  package Float_IO is new Text_IO.Float_IO( Float );
-- remaining declarations and code is the same with references
-- to Stack_Package replaced by Complex_Stack,
-- e.g. Complex_Stack.Push( Complex_Value, Stack );
```

Listing 4

```
generic
  type Float_Type is digits<>;
package Complex_Number_Package is
  Complex_Numeric_Error : exception;
  type Complex_Number is private;
  ...
end Complex_Number_Package;
package body Complex_Number_Package is
...
  function "+"( First_Number  : in Complex_Number;
                Second_Number : in Complex_Number )
    return Complex_Number is
    New_Number : Complex_Number;
  begin  -- "+"
    New_Number.Real_Part := First_Number.Real_Part +
                            Second_Number.Real_Part;
    New_Number.Imag_Part := First_Number.Imag_Part +
                            Second_Number.Imag_Part;
    return New_Number;
  exception
    when Numeric_Error => raise Complex_Numeric_Error;
  end "+";
```

```
-- The functions "-" and "*" are similar to "+";
  ...
end Complex_Number_Package;

generic
  type Item is private;
package Stack_Package is
  Stack_Overflow  : exception;
  Stack_Underflow : exception;
  ...

end Stack_Package;

package body Stack_Package is
...
  procedure Pop( Element :    out Item;
                 Stack   : in out Stack_Type ) is
  begin  -- Pop
    Element := Stack.Info( Stack.Top );
    Stack.Top := Stack.Top - 1;
  exception
    when Constraint_Error =>
      raise Stack_Underflow;
  end Pop;

  procedure Push( Element : in     Item;
                  Stack   : in out Stack_Type ) is
  begin  -- Push
    Stack.Top := Stack.Top + 1;
    Stack.Info( Stack.Top ) := Element;
  exception
    when Constraint_Error =>
      raise Stack_Overflow;
  end Push;
end Stack_Package;
  with Complex_Number_Package;
  with Stack_Package;
  with Text_IO;
procedure Calculator is
  package Complex_Package is new Complex_Number_Package( Float );
  use Complex_Package;
  package Complex_Stack is new Stack_Package( Complex_Number );
  package Float_IO is new Text_IO.Float_IO( Float );
...

Ill_Formed_Expression : exception;
Syntax_Error          : exception;
Complex_Data_Error    : exception;

procedure Read_Complex( Number : out Complex_Package.Complex_Number ) is

  Real_Part, Imag_Part : Float;
  Delimiter : Character;
```

```
begin  -- Read_Complex
  Text_IO.Get( Delimiter );   Float_IO.Get( Real_Part );
  Float_IO.Get( Imag_Part );  Text_IO.Get( Delimiter );
  Number := Complex_Package.Create( Real_Part, Imag_Part );
exception
  when others =>
    raise Complex_Data_Error;
end Read_Complex;
...

begin  -- Calculator
  Evaluate_All_Expressions:
  while not Text_IO.End_Of_File loop
    Evaluation_Block:
    begin
      Text_IO.Get( Next_Character );
      Evaluate_One_Expression:
      while Next_Character /= '=' loop
        case Next_Character is
          -- code is as above with references
          -- to Stack_Package replaced by Complex_Stack,
          -- e.g. Complex_Stack.Push( Complex_Value, Stack );
        when ' ' =>
          null;
        when others =>
          raise Syntax_Error;
        end case;
        Text_IO.Get( Next_Character );
      end loop Evaluate_One_Expression;
      Complex_Stack.Pop( Result, Stack );
      Write_Complex( Result );
      if not Complex_Stack.Empty( Stack ) then
        raise Ill_Formed_Expression;
      end if;
    exception
      when Complex_Stack.Stack_Underflow | Ill_Formed_Expression =>
        Text_IO.Put_Line( "Ill formed expression." );
      when Complex_Stack.Stack_Overflow =>
        Text_IO.Put_Line( "Expression too complex." );
      when Complex_Package.Complex_Numeric_Error =>
        Text_IO.Put_Line( "Numeric Error." );
      when Complex_Data_Error =>
        Text_IO.Put_Line( "Invalid form for complex number." );
      when Syntax_Error =>
        Text_IO.Put_Line( "Invalid syntax." );
    end Evaluation_Block;
    Complex_Stack.Clear( Stack );
    while Next_Character /= '=' loop
      Text_IO.Get ( Next_Character );
    end loop;
  end loop Evaluate_All_Expressions;
exception
  when Text_IO.End_Error => null;
end Calculator;
```

Listing 5

```
  with Text_IO;
  with Stack_Package;
  with Complex_Number_Package;
procedure Concurrent_Calculator is
  package Complex_Package is new Complex_Number_Package( Float );
  use Complex_Package;
  type Tag_Type is ( Number, Pending );
  type Operand_Type ( Tag : Tag_Type := Number ) is
    record
      case Tag is
        when Number =>
          Value : Complex_Number;
        when Pending =>
          null;
      end case;
    end record;

  package Operand_Stack is new Stack_Package( Operand_Type );
  package Float_IO is new Text_IO.Float_IO( Float );
  Stack : Operand_Stack.Stack_Type;
  ...

  task Arithmetic_Unit is
    entry Clear_Unit;
    entry Add( Operand1 : in Complex_Number;
               Operand2 : in Complex_Number );
    entry Subtract( Operand1 : in Complex_Number;
                    Operand2 : in Complex_Number );
    entry Multiply( Operand1 : in Complex_Number;
                    Operand2 : in Complex_Number );
    entry Get_Result( Result_Value : out Complex_Number );
  end Arithmetic_Unit;

  task body Arithmetic_Unit is
    package Complex_Stack is new Stack_Package( Complex_Number );
    Result_Stack : Complex_Stack.Stack_Type;
    Result_Value, Local1, Local2 : Complex_Number;
  begin
    loop
      select
        accept Clear_Unit;
        Complex_Stack.Clear( Result_Stack );
      or
        accept Add( Operand1 : in Complex_Number;
                    Operand2 : in Complex_Number )
        do
          Local1 := Operand1;   Local2 := Operand2;
        end Add;
        Result_Value := Local1 + Local2;
        Complex_Stack.Push( Result_Value, Result_Stack );
      or
```

```
        accept Subtract( Operand1 : in Complex_Number;
                         Operand2 : in Complex_Number )
        do
          ...
      or
        accept Multiply( Operand1 : in Complex_Number;
                         Operand2 : in Complex_Number )
        do
          ...
    or
        accept Get_Result( Result_Value : out Complex_Number )
        do
          Complex_Stack.Pop( Result_Value, Result_Stack );
        end Get_Result;
      or
        terminate;
      end select;
    end loop;
  end Arithmetic_Unit;

  procedure Pop( Result_Value : out    Complex_Number;
                 Stack        : in out Operand_Stack.Stack_Type ) is

    Stack_Item : Operand_Type;
  begin -- Pop
    Operand_Stack.Pop( Stack_Item, Stack );
    if Stack_Item.Tag = Pending then
      Arithmetic_Unit.Get_Result( Result_Value );
    else
      Result_Value := Stack_Item.Value;
    end if;
  end Pop;

  procedure Push( Complex_Value : in Complex_Number;
                  Stack         : in out Operand_Stack.Stack_Type ) is

  begin  -- Push
    Operand_Stack.Push(( Tag => Number, Value => Complex_Value),
                         Stack );
  end Push;

  procedure Push_Token( Stack : in out Operand_Stack.Stack_Type ) is

  begin
    Operand_Stack.Push(( Tag => Pending ), Stack );
  end Push_Token;
  ...

begin  -- Calculator
  Evaluate_All_Expressions:
  while not Text_IO.End_Of_File loop
    Evaluation_Block:
    begin
      Text_IO.Get( Next_Character );
      Evaluate_One_Expression:
      while Next_Character /= '=' loop
        case Next_Character is
          when 'E' =>
            Read_Complex( Complex_Value );
            Push( Complex_Value, Stack );
          when '+' | '-' | '*' =>
            Pop( Right_Operand, Stack );
            Pop( Left_Operand, Stack );
            Push_Token( Stack );
            case Next_Character is
              when '+' =>
                Arithmetic_Unit.Add( Left_Operand, Right_Operand );
              when '-' =>
                Arithmetic_Unit.Subtract( Left_Operand,
                                          Right_Operand );
              when '*' =>
                Arithmetic_Unit.Multiply( Left_Operand,
                                          Right_Operand );
              when others =>
                null;
            end case;
          when ' ' =>
            null;
          when others =>
            raise Syntax_Error;
        end case;
        Text_IO.Get( Next_Character );
      end loop Evaluate_One_Expression;
      Pop( Result_Value, Stack );
      Write_Complex( Result_Value );
      if not Operand_Stack.Empty( Stack ) then
        raise Ill_Formed_Expression;
      end if;
    exception
    ...
    end Evaluation_Block;
    Operand_Stack.Clear( Stack );
    Arithmetic_Unit.Clear_Unit;
    while Next_Character /= '=' loop
      Text_IO.Get ( Next_Character );
    end loop;
  end loop Evaluate_All_Expressions;
exception
  when Text_IO.End_Error =>
    null;
end Concurrent_Calculator;
```

Listing 6

```
  with Text_IO;
  with Stack_Package;
  with Complex_Number_Package;
procedure Concurrent_Calculator is
```

```
package Complex_Package is new Complex_Number_Package( Float );
use Complex_Package;
package Float_IO is new Text_IO.Float_IO( Float );
type Operation_Type is ( None, Addition, Subtraction, Multiplication );
task type Adder_Type is
  entry Operate( Operand1 : in Complex_Number;
                 Operand2 : in Complex_Number );
  entry Get_Result( Result : out Complex_Number );
end Adder_Type;
-- Subtractor_Type and Multiplier_Type are specified similarly.
...

type A_U_Type( Operation : Operation_Type ) is
  record
    case Operation is
      when None           => Register   : Complex_Number;
      when Addition       => Adder      : Adder_Type;
      when Subtraction    => Subtractor : Subtractor_Type;
      when Multiplication => Multiplier : Multiplier_Type;
    end case;
  end record;
type A_U_Access_Type is access A_U_Type;
package Stacks is new Stack_Package( A_U_Access_Type );
Complex_Zero : constant Complex_Number := Create( 0.0, 0.0 );
...

task body Adder_Type is
  Local_Result, Local1, Local2 : Complex_Number;
begin
  accept Operate( Operand1 : in Complex_Number;
                  Operand2 : in Complex_Number )
  do
    Local1 := Operand1;  Local2 := Operand2;
  end Operate;
  Local_Result := Local1 + Local2;
  accept Get_Result ( Result : out Complex_Number )
  do
    Result := Local_Result;
  end Get_Result;
end Adder_Type;

-- The bodies of Subtractor_Type and Multiplier_Type
-- are similar.
...

procedure Pop( Result  :    out Complex_Number;
               Stack   : in out Stacks.Stack_Type ) is
  A_U_Access : A_U_Access_Type;

begin -- Pop
  Stacks.Pop( A_U_Access, Stack );
  case A_U_Access.Operation is
    when None =>
      Result := A_U_Access.Register;
      when Addition =>
        A_U_Access.Adder.Get_Result( Result );
      when Subtraction =>
        A_U_Access.Subtractor.Get_Result( Result );
      when Multiplication =>
        A_U_Access.Multiplier.Get_Result( Result );
    end case;
  end Pop;

  procedure Push( Operation : in     Operation_Type;
                  Left_Op   :        Complex_Number;
                  Right_Op  :        Complex_Number := Complex_Zero;
                  Stack     : in out Stacks.Stack_Type ) is
    A_U_Access : A_U_Access_Type;

  begin
    A_U_Access  := new A_U_Type( Operation );
    case A_U_Access.Operation is
      when None =>
        A_U_Access.Register := Left_Op;
      when Addition =>
        A_U_Access.Adder.Operate( Right_Op, Left_Op );
      when Subtraction =>
        A_U_Access.Subtractor.Operate( Right_Op, Left_Op );
      when Multiplication =>
        A_U_Access.Multiplier.Operate( Right_Op, Left_Op );
    end case;
    Stacks.Push( A_U_Access , Stack );
  end Push;
  ...
begin  -- Calculator
  Evaluate_All_Expressions:
  while not Text_IO.End_Of_File loop
    Evaluation_Block:
    begin
      Text_IO.Get( Next_Character );
      Evaluate_One_Expression:
      while Next_Character /= '=' loop
        case Next_Character is
          when 'E' =>
            Read_Complex( Complex_Value );
            Push( None, Complex_Value, Stack => Stack );
          when '+' | '-' | '*' =>
            Pop( Right_Op, Stack );
            Pop( Left_Op, Stack );
            case Next_Character is
              when '+' =>
                Push( Addition, Left_Op, Right_Op, Stack );
              when '-' =>
                Push( Subtraction, Left_Op, Right_Op, Stack );
              when '*' =>
                Push( Multiplication, Left_Op, Right_Op, Stack );
              when others =>
                null;
            end case;
          when ' ' =>
            null;
          when others =>
            raise Syntax_Error;
        end case;
        Text_IO.Get( Next_Character );
      end loop Evaluate_One_Expression;
      Pop( Result, Stack );
      Write_Complex( Result );
      if not Stacks.Empty( Stack ) then
        raise Ill_Formed_Expression;
      end if;
    exception
    ...

    end Evaluation_Block;
    Stacks.Clear( Stack );
    while Next_Character /= '=' loop
      Text_IO.Get( Next_Character );
    end loop;
  end loop Evaluate_All_Expressions;
exception
  when Text_IO.End_Error =>
    null;
end Concurrent_Calculator;
```

Building an Ada Organization: AdaKeypers and Their Roles in the Project Training Plan

By Eileen S.Quann
Fastrak Training Inc.
9175 Guilford Road, Suite 300
Columbia, Maryland 21046-1802

Abstract

Finding yourself suddenly responsible for all or part of an Ada development effort can be an intimidating experience. Rarely are there enough, if any, peopl` with Ada experience available for your project. Often, you need them trained immediately. The managers who are responsible for delivering the project on time and at cost are concerned that using Ada will jeopardize those objectives. Management's lack of Ada experience, coupled with limited or nonexistent Ada experience of their staff, raises the question of whether anyone will be there to provide help and guidance when it is needed.

Often the first step is to schedule Ada language training for the programmers. The real solution, however, must address the broader issue of developing an Ada organization. This paper identifies the critical components of developing a strong Ada organization and discusses two of those components: the Ada key personnel (AdaKeypers) and the Project Training Plan. The AdaKeypers are the project personnel responsible for transitioning their organization into the Ada era. The Project Training Plan is the road map for that transition.

Traditionally, the role of the trainer has ended when the student leaves the classroom. This paper extends the concept of trainer to include longer term project support for design reviews, software inspections, documentation reviews, etc., until the project personnel feel comfortable with their own level of Ada expertise. This paper outlines what needs to be taught and when, and describes how you would go about developing a comprehensive training plan

that integrates Ada with your company's way of doing business, whether you are building an Ada organization of 10 people or 1000.

Building an Ada Organization Starts With Good Project Documentation

Early in the project life (or sooner if possible), it is necessary to define and then document the standards, plans and procedures that will guide the project throughout its life cycle. Most large companies and many smaller ones have corporate standards, policies, and procedures that are applied to all development projects. Such standards are essential to the success of Ada projects, but only after they have been modified to accommodate the unique requirements of Ada. Because of the rapid evolution of the Ada environment, most plans and procedures written more than a couple of years ago are largely obsolete. Failure to modify them may produce worse results than using no standards at all. Not intended to be static documents, they must form the baseline for future evolution.

The Program Management Plan (PMP) forms the basis for direction within the project. It defines the project organization and the responsibilities of key personnel. It documents the project schedule, defines the staffing approach and staffing plans and addresses the project risks. All other plans must be consistent with the direction provided in the Program Management Plan. The Software Development Plan, the Software Standards and Procedures, the Project Training Plan, the project development methodology, the use and tailoring of MIL-STD 2167a, and 2168 where applicable, Program Design Language and Coding Guidelines, Quality Assurance and Configuration Management Plans all need to address and become part of the Ada development environment.

AdaKeypers are the Building Blocks of a Strong Ada Organization

Having identified the initial need for appropriately tailored project documentation, who will develop it? In building any organization, it is necessary to identify the key personnel who will guide the program as it develops. Given the many functional organizations in a project and the need for strong communication links between those organizations, key people need to come from all areas of the project. I have coined the term AdaKeypers to refer to this group of key personnel on an Ada project.

The AdaKeypers are the building blocks, the foundation from which the Ada project expertise will evolve. At the start of the project, the AdaKeypers may or may not have Ada experience, though some experience is certainly desirable. AdaKeypers must include people from technical management, system engineering, software engineering, and quality assurance. Depending on the structure of your organization, other groups may also need to be represented. The mission of the AdaKeypers is to become the Ada experts, each in some area of specialization. They will often need outside consulting and training assistance in the beginning of the project and through the early transition stages. AdaKeypers will play lead roles in the development of much of the project documentation.

AdaKeypers From All Areas of the Organization Lead the Transition to Ada

We have traditionally accepted the concept of senior technical staff members, "gurus" of a project, who provide expert technical leadership and guidance. AdaKeypers extend that concept to other areas of an organization where Ada expertise is needed.

It is important to have one or more software managers recognized as AdaKeypers. These people need to understand the impact of Ada on the management issues in order to write plans, provide input to higher management and review development products. Building on their management experience, their involvement is necessary to develop standards and procedures that are consistent with Ada, to promote sound software engineering principles, and to follow corporate policies. They must have a sufficiently senior position to have easy access to the project manager and must be able to present Ada issues at management meetings. Their technical backgrounds must be strong enough to enable them to integrate technical application issues with Ada and to evaluate the quality of software products. Software managers in the AdaKeypers group can ensure that Ada issues get the visibility and attention of higher management.

The need for AdaKeypers among the software staff is obvious. Whether they are the lead software designers or language specialists, they serve a critical role on the project. In addition to providing technical leadership, they can advise on alternative Ada designs, assist in the resolution of compiler and other tool problems, and answer technical Ada questions. Their participation in design and code reviews can promote good Ada software while supplementing the Ada training for the project developers.

Unfortunately, on many projects, the system engineering organization knows very little about Ada. It is important to include AdaKeypers from this group.

Without a knowledge of Ada, system architects and designers, experienced in procedural methodologies, often create a design that does not incorporate the rich features and capabilities of Ada and may even be awkward to implement in Ada. When software specifications are produced by a system engineering group unfamiliar with Ada, the functional decomposition of requirements may add unnecessary re-work for a good Ada implementation.

An often overlooked group that needs to be included in the AdaKeypers program is Quality Assurance. To be effective, the quality assurance staff must have strong software backgrounds and good training in Ada. They can then play a major role in developing project standards and procedures. They can participate in design and code reviews. Eventually, as their expertise grows, they can assume the role of trainers for the project. Quality Assurance personnel must work closely with the development group throughout the design and implementation phases to answer questions, provide guidance, and monitor adherence to project standards and procedures. A Quality Assurance organization is functioning at its best when all project members use them as a resource to ensure timely delivery of quality products. This will not happen unless the quality assurance staff are knowledgeable, well trained in Ada, and willing to work with the staff.

Recognition of the AdaKeypers' Roles Can Speed up the Transition

When project personnel know who to go to for help, less time is wasted searching for answers. A well-defined, recognized group of people, each with specialized knowledge of different aspects of the project, can work together to coordinate the transition. AdaKeypers should have special training and should be involved early in the project life cycle developing project documentation. Working closely with one another and with outside consultants, AdaKeypers can both speed up and smooth the organization's transition to Ada. Special recognition, in the form of certificates, or logos on their door, can identify these people quickly to new project personnel.

AdaKeypers Are Strong Members of the Software Design Review Board

A Software Design Review Board can be an essential component in building an Ada organization, particularly for a large project. The Software Design Review Board is responsible for reviewing and approving top level designs, formulating technical guidance pertaining to software issues and approving each use of certain Ada features. To prevent run time problems, project standards often limit the Ada constructs, such as tasking or generics, that programmers may use. To provide needed flexibility and encourage programmer creativity, while still safeguarding project objectives, developers can submit requests to the Software Design Review Board for authorization to use those features. The Board can ensure that project standards are equitably and consistently applied across the project, designs are carefully reviewed and unnecessary re-work is avoided. AdaKeypers are strong members on this Board.

AdaKeypers Must have Training and Support

AdaKeypers need training in the technical and management issues that must be considered in developing the project plans, standards and procedures. Outside consultants can provide the training and assist in the development of those plans, but unless project plans have actually been developed by project personnel, sponsorship of those plans is difficult to obtain. People support that which they help to create. Each organization has its own culture, its own way of doing business, its 'corporate personality'. Rarely can outside consultants capture that, or incorporate it into project plans. As a result, plans developed by outside groups are frequently ignored or misunderstood, because they fail to reflect the way work will actually be performed. Consultants, however, can be extremely useful in providing support and guidance to AdaKeypers during the development of the plans, and later, during project implementation, until AdaKeypers feel comfortable with their own level of expertise.

The Project Training Plan is the Road Map for the Transition

The Project Training Plan needs to address the project specific training requirements as well as Ada training needs. Project specific training includes project orientation, covering such topics as project purpose, scope, organization, schedules, deliverables; in depth application training; and training in project policies, standards and procedures. Ada training should include courses in management issues, design methodology, language (features, coding, compiling, testing), Program Design Language (PDL), and, if not covered in standards and procedures, design review and inspection procedures.

The Project Training Plan should list and describe all courses, with course outlines, course pre-requisites, and the intended audience. Project personnel should know what training they will receive and approximately when they will receive it. Some courses may be as short as three hours, others may last over a week. It is recommended that courses which exceed a week be partitioned into one week segments so that students will have the opportunity to assimilate the material and perhaps use it, before moving on to increasingly complex topics.

The transition of Ada expertise from outside trainers or consultants to AdaKeypers to all project personnel can be described in the Project Training Plan.

The following concepts, when incorporated into your plan will ensure an orderly and smooth transition to Ada.

Provide training that is tailored to your project's way of doing business. Don't expect off-the-shelf training to provide all the material your project needs. Take advantage of available material (don't re-invent the wheel), but don't rely on it completely.

Every person on the project, including administrative support, should receive some mandatory Ada training. Where possible, include government monitors in your training classes. They are much more likely to support your efforts if they have had the same training and understand what you are doing and how you are doing it.

Allow time in the project schedule for training. Often, training overlaps other project activities and students are torn between attending class and completing deliverables. Training can be expensive. Don't dilute its effectiveness.

Provide the training when people need the information. Providing training either too early, or too late in a project reduces its effectiveness. Classes in management issues, software engineering, and project standards and procedures should always be given early in the project, since these concepts take time to absorb and the objective of these classes is to change the way people think about the development process. Classes in language features and design should be given early in the project so the students can have the opportunity to use what they have learned by developing software tools and prototypes. PDL and language implementation classes should be offered at the start of design. Classes on testing issues can be addressed just before that phase of the development begin. Providing the right information at the right time ensures a higher level of retention.

Training Doesn't End When the Student Leaves the Classroom

Traditionally the role of the trainer has ended when the student leaves the classroom. Because of the newness and complexity of Ada, it is desirable for project staff to have continuous access to skilled Ada personnel. Outside consultants who provide the initial training can fill this role at the beginning of the project. Over time, and with support, the AdaKeypers can assume more responsibility as their Ada expertise grows. The objective is for the AdaKeypers to evolve into the project trainers and Ada experts.

AdaKeypers play a critical role in building an Ada organization. They form a team which represents each organization participating in the project. With proper training and management support, AdaKeypers will work together to ensure not only adequate training for project personnel, but ongoing support throughout the project life cycle.

This Page Left Blank Intentionally

"Curing the Turbonic Plague"

by
Colen K. Willis
Major, U.S. Army
Assistant Professor of Computer Science
United States Military Academy

Q: If I understand this correctly, you're going to interview yourself?
A: That's correct.

Q: Why?
A: Well, I've tried several ways to present my ideas but none of those ways seemed to work. I saw this method used before and it worked well.

Q: What do you want to talk about?
A: I'd like to offer thoughts on Ada education ... mainly on my experience teaching Ada at the United States Military Academy.

Q: Why?
A: I'm not sure. I'm hesitant because what I have to offer is not based on scientific study. I just feel compelled to offer something.

Q: But you must have something on your mind.
A: Well, I've just spent five years of my life studying and teaching Software Engineering with Ada. As an Army officer, I'm about ready to leave my teaching position in the computer science department at the U.S. Military Academy and return to the "real" Army. I've been carrying around lots of thoughts on how to teach Ada and this may be my only chance to unload them.

Q: OK. Where do you want to start?
A: Well, I could give a review of Ada at West Point since we began teaching it in 1979.

Q: Oh, please ...
A: Don't worry ... I won't. I want to focus only on the approach we used to teach Ada this semester at West Point.

Q: OK, but the title of this paper is "Curing the Turbonic Plague". I thought you wanted to talk about Ada education not medicine.
A: I do.

Q: Then why that title?
A: Two reasons. First, the "Turbonic Plague" describes a sort of disease that is a real concern of mine. And second,

the "curing" of that disease describes our approach to teaching Ada.

Q: At the possibility of sounding stupid, what is the "Turbonic Plague"?
A: I couldn't wait till you asked that question. I assume you're aware of the popular micro-based programming systems, called "Turbo" or "Quick" systems.

Q: Humor me.
A: Well, these systems are easy-to-use, micro-based programming environments that provide "lightning-fast" compilation. The core of these systems is a full-screen editor from which the programmer can easily invoke the compiler or file manager.

Q: That doesn't sound so unique. A lot of programming systems do that.
A: True, but what makes the "Turbo" system unique is its lightning-fast compilation speed.

Q: How fast is lightning-fast?
A: Fast! Sometimes it appears instantaneous! That's where the "Turbo" comes from. Remember though, this isn't a scientific study...I didn't perform any benchmark tests to get exact compilation times. So just let me leave it at that ... "Turbo" is fast!

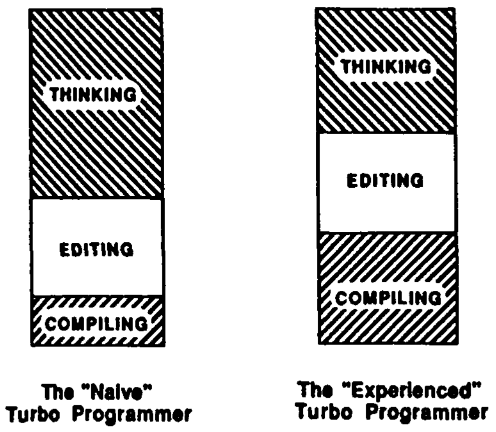
Q: I believe you. But how can it be so fast?
A: Well, besides being well written, "Turbo" systems are designed to find the <u>first</u> compilation error then terminate the compilation and return the user to the editor at the position in the source code where the error occurred.

Q: Wow, that's pretty impressive. Who would use one of these "Turbo" systems?
A: Well, I can't speak for programmers in industry because I've never been out there. But I can tell you that "Turbo" systems are used extensively in undergraduate courses throughout America - we use "Turbo Pascal" by Borland at West Point. I can also tell you that "Turbo" systems are owned and used by lots of amateur programmers.

Q: So what's this about the "Turbonic Plague"?
A: Well, I believe that "Turbo" languages, while being <u>fast</u> and <u>easy-to-use</u>, have had a very harmful effect on computer programming and on our approach to teaching programming. I refer to this harmful effect as the "Turbonic Plague".

Q: Do you mean to tell me there is something wrong with fast, easy-to-use compilers?
A: Well, it's more than that. I'm all for fast compilers and nicely integrated programming support systems. But I've noticed that these systems seem to promote bad programming

habits. Let me show you a graphic that I've created to summarize my experience with "Turbo" systems. This is a "gut feeling" chart ... but I suspect that most other introductory level programming instructors would agree with what it illustrates.



The Metamorphosis of "Turbo" Programmers
(Figure 1.)

Q: What do the bars represent?
A: The bar chart on the left shows how the "naive" programmer - one who is studying programming at the introductory level - would likely allot time to completing a programming assignment. I've noticed that the naive programmer spends a fair amount of time in the early phase of the program development process THINKING! Part of that thinking time is forced (as in most introductory programming courses, we stress programming logic and structured design) and the remaining part is self-induced. Using a "Turbo" system for program development, the naive programmer spends a fair chunk of time in the editor - primarily learning its features. This disciplined approach to the program development process, coupled with the lightning-fast compilation speed, result in an extremely small amount of time being committed to compilation.

Q: OK, what about the other bar?
A: The other bar shows how an "experienced" user allots time in the programming process. Do you notice anything peculiar when you compare the two bars?

Q: It appears that the amount of time devoted to compiling has increased, the amount of time devoted to thinking has decreased and the amount of time dedicated to editing has remained the same.
A: Exactly.

Q: Explain!
A: My experience shows that "Turbo" users - enamored by fast compilation speeds - eventually begin to devote less and less time to "up-front thinking". Instead, experienced "Turbo" users tend to think as they program ... in much the same way that a laboratory rat negotiates a maze. Rather than thinking logically about how to attack a problem, the experienced "Turbo" programmer - attracted by the ease and speed of the "Turbo" environment - often bypasses the planning phase and immediately begins to code. Like the rat that takes an "unthinking" path, then hits a "wall", corrects his course, hits another "wall", corrects, and so on ... the experienced "Turbo" programmer builds an "unthinking" program, finds an error (compiles), corrects the error (edits), finds another error, corrects the error, and so on.

Q: So your claim is that "Turbo" promotes thinking "on-the-fly".
A: Exactly!

Q: But isn't thinking "on-the-fly" still thinking?
A: Maybe. But I contend that this sort of thinking is less desirable than more methodical, deliberate, "up-front thinking".

Q: How so?
A: Well, it seems to me that "up-front thinking" is directed toward a macro view of the problem at hand while "on-the-fly-thinking" is more likely to focus on a collection of micro-views of the problem. The best programmers are those who have a good grasp of the problem and have formulated a broad, macro plan before they begin coding. I also contend that the quality of software - especially large, complex software - is enhanced when the programmer begins with a well-defined macro view of the problem.

Q: Makes sense. How does this metamorphosis from "up-front thinker" to "on-the-fly-thinker" occur?
A: I believe it's due to the addictive nature of "Turbo" systems. These systems are so fast and so "user sympathetic" that they give the impression that they will do all phases of programming -including the thinking phase - for the user. What happens is that a student who was trained using the proper "Ready, Aim, Fire" approach to programming eventually employs a "Ready, Fire, Aim" approach!

Q: Good analogy. You have mentioned "Turbo" Pascal several times. Are there other languages integrated into "Turbo" systems?
A: I know of "Turbo" C, "Turbo" Basic and "Turbo" Prologue - there are probably others. What's interesting is that these

"Turbo" environments really distort the underlying languages with extensions and shortcuts.

Q: Are you saying that "Turbo" Pascal may not look like other versions of Pascal?
A: Exactly. At this point I suppose we could discuss the broader topic of language standardization...

Q: No, I get the point.
A: I guess what really concerns me about "Turbo" systems - from the language stand point - is that they are often used because of their appealing environment ... not for the virtues of the underlying programming language.

Q: What do you mean?
A: Well, computer programming is offered in lots of flavors ... from introductory programming courses to courses in software design and engineering. Pascal, for example, is an appropriate choice of languages at the introductory level for teaching structured programming and "programming-in-the-small" - especially for those who are taking programming as a required course and who will likely never have to write a computer program after the first semester of their sophomore year in college!

Q: Keep going.
A: It's my belief, however, that courses in software design and engineering should employ more appropriate languages - like Ada - that better support software engineering ideas.

Q: Sounds logical.
A: It sounds logical. But what really happens is that we often forego language appropriateness in an effort to use the fastest compiler and most convenient programming environment. The result ... we end up inappropriately teaching "Programming in the Large" using "Turbo" Pascal!

Q: So what conclusions do you draw from this study of "Turbo" systems?
A: I have drawn two conclusions. First, I believe that these systems, because of their lightning-fast compilers, easy-to-use environments and the way in which they distort the underlying programming language, promote bad programming practices and harm the process of learning computer programming.

Q: OK, and the second conclusion?
A: "Turbo" systems have a lot of good features. First, the idea of having a set of programming tools available on a personalized microcomputer is obviously appealing. Second, the "Turbo" system offers a "comfortable" user interface ... not too difficult to learn and use but, at the same time, powerful enough to provide proper support. And lastly, I noticed that "Turbo" programmers always have a smile on

their face ... sometimes innocent, sometimes devilish ... but always a smile. "Turbo" systems are FUN!

Q: So you're not completely against "Turbo" systems.
A: Not at all. "Turbo" systems may well be appropriate for "programming-in-the-small", prototyping and so on. I am concerned, however, about their impact on teaching good software engineering, design and programming of large, complex software systems. The undisciplined use of "Turbo" systems has generated an epidemic of "unthinking" programmers and I don't think the situation is going to get any better unless we act now.

Q: OK, ready for the big question? How do we stop the "Turbonic Plague"?
A: Well, there are probably lots of ways to stop it. I considered initiating a "Say No to Turbo" campaign in which I would issue T-shirts with a "Turbo Buster" logo to anyone who joined the campaign.

Q: Any luck?
A: Nope, too passive. I also considered a plan for building a "Turbo Rehabilitation Center" in every major university town in America.

Q: And?
A: Too little too late! I then considered preparing a proposal to the Department of Defense requesting funds to support research in developing a "Turbo vaccine". I gave up in light of Graham-Rudmann!

Q: Did you ever find a cure?
A: Well, I believe I've taken a major step in the right direction.

Q: Let's hear about it.
A: OK, I guess I can best introduce my cure by paraphrasing the Surgeon General of the United States...

> "The only known prevention of the [Turbonic Plague], barring abstinence, is the use of [ACTIVE Ada]"

Q: So you refer to your cure as "ACTIVE Ada"?
A: Right.

Q: Tell me about it.
A: "ACTIVE Ada" is a teaching philosophy designed to produce "thinking" software engineers, designers, and programmers. The philosophy employs a customized collection of microcomputer programming support tools - tools that actively promote learning! The most important tool in this set is the Ada programming language.

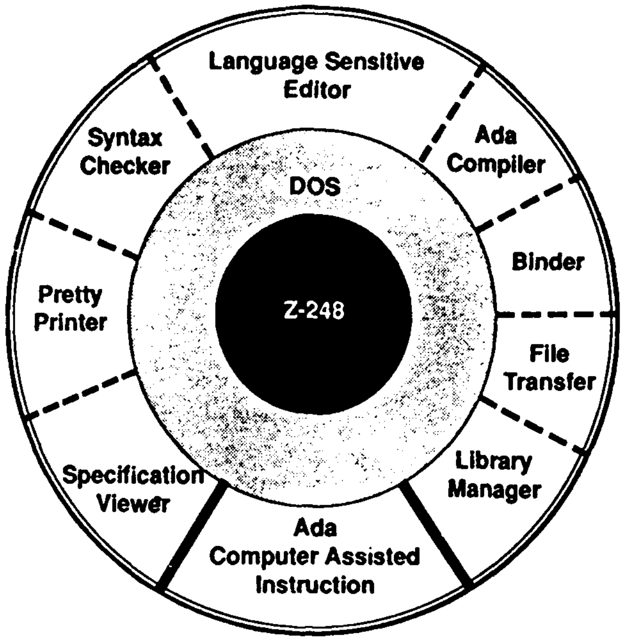
Q: Sounds interesting! How did you develop this cure?
A: Two years ago, we began teaching "Turbo" Pascal in our mandatory, freshmen-level "Introduction to Computers and Pascal Programming" course. I noticed lots of "rat-like" programming by introductory level programmers about midway through the semester!

Q: Keep going.
A: "Turbo" Pascal has also become the basic programming language for the computer science program at West Point. During a cadet's sophomore year, he or she formally begins the computer science program by taking "Structured Programming" and "Data Structures and Algorithms" - both Pascal-based. More serious symptoms of the "Turbonic Plague" begin to appear in computer science concentrators at that time.

Q: I can see the headlines now ... "The Turbonic Plague Hits West Point".
A: Well, it occurred to me as I prepared material for my second semester course, "Software Engineering with Ada", that I had a perfect opportunity to cure the "plague". I saw three immediate "targets". First, the cadets were being issued personal micro-computers (Zenith 248's). Second, micro-based Ada programming support tools were beginning to "hit the street". And lastly, with the expansion of our micro-computing facilities, I saw an opportunity to present Ada - ACTIVELY - in a laboratory setting.

Q: Why don't you explain each of those "targets"?
A: OK. First, the issuing of Zenith 248's to cadets means that we can make computer science education much more personal. Cadets are issued "Turbo" Pascal along with their computers so that, after the introductory programming course, they are fairly familiar with an integrated programming environment. My objective then was to make ACTIVE use of their personal computers and to build on their familiarity with a micro-based programming support environment.

Q: OK, what about the second "target"?
A: Well, I noticed through advertisements in computing periodicals, telephone calls from vendors and advice from friends that micro-based Ada programming support tools were beginning to show up on the market. At that time there were two AT-class validated compilers and several rather primitive environments. I analyzed very carefully what micro-based tools would not only support teaching Ada ... but would actively teach Ada. I developed a model for a collection of tools that I called the MATSE - Micro-Ada Training Support Environment. The next diagram graphically portrays this set of tools.



The Micro-Ada Training Support Environment (MATSE)
(Figure 2.)

Q: Interesting. Can you explain the diagram?
A: Sure. For those familiar with the Ada effort, this looks very similar to the Ada Programming Support Environment (APSE) concept. At the core of the MATSE you notice the "iron" - the Z-248 - and the DOS operating system. My idea was to make this core as transparent as possible. Around that core sits an integrated layer of Ada TRAINING support tools.

Q: So the user works with the tools in the outer layer and never has to be concerned about the hardware or operating system.
A: Exactly. This is a lot like a "Turbo" environment in terms of "ease of use" and is important in teaching Ada for two reasons. First, Ada is "bigger" than traditional instructional languages like Pascal. If we have to use a collection of independent and autonomous tools to build Ada systems, then programming becomes more difficult and teaching Ada becomes even more difficult. If you've ever programmed on systems where you move from an independent editor, to the operating system, to the compiler, back to the operating system, to the linker and so on, then you know what I mean.

Q: I'm not a programmer but it does sound frustrating!
A: Secondly, at West Point, we'll likely continue using Pascal as the basic programming language in the introductory programming course for some time. (If I didn't mention it already, these "Turbo" systems are very affordable!) The transition of using Ada by our computer science majors would be greatly enhanced if the Ada language was available from an environment similar in use to the "Turbo" Pascal environment.

Q: Sounds logical. Explain some of the tools in the MATSE.
A: OK, the "hub" of the system is the editor. I was looking for more than a full-screen editor ... I was interested in an Ada language-sensitive editor that would ACTIVELY teach Ada by assisting the programmer in building syntactically correct programs. This would allow us to worry less about syntax and focus more on design and semantics. Notice also that the MATSE specification includes a syntax checker that would perform rapid syntax checking so that the programmer could perform fewer complete compilations.

Q: What do you mean, the editor is the "hub"?
A: Well, we need to "hang around" where the action is. Sort of like the kitchen in my house! When the user enters the environment, he or she is immediately "placed" in the editor. From the editor, it's important that the user have access (in most cases, direct access) to all of the other tools.

Q: So, for example, you can invoke the compiler directly from the editor.
A: Exactly ... and when compilation is complete, you are returned automatically back to the editor.

Q: Tell me more about the compiler.
A: Three points. First, I specified in the MATSE model that the compiler must support <u>all</u> aspects of Ada <u>correctly</u>. As my Mom says ... "Do it right or don't do it at all!". Secondly, I was interested in a compiler that captured as many errors as possible ... to discourage "rat-like" behavior. And lastly, I was looking for a compiler that was ACTIVE in how it taught Ada. In other words, it had to have helpful error messages.

Q: And obviously, it had to be fast!
A: NO! Remember that my complaint about "Turbo" environments is that they might be <u>too</u> fast and thereby tend to discourage "thinking". Pascal is designed - like a small sports car - to be fast. Ada, on the other hand, is designed like a semi-trailer truck ... the focus is not on speed but on efficient and powerful load carrying over a long haul.

Q: Are you saying that we will never have lightning-fast Ada compilers?
A: I'm saying that compilation speed is not the overriding factor in the design of Ada compilers. There are a lot of other considerations to be taken in the Ada compiler writing business that are, in the end, far more important than compiler speed to building well-engineered software.

Q: Enough about the compiler ... tell me about the other tools. What is the "library manager"?
A: Well, this is one of the keys to studying Ada in a software engineering context. It is the tool that manages all of the parts (compilation units) of an Ada program. This tool can become very sophisticated ... to the point that it actually hinders the learning process. To support my idea of ACTIVELY teaching Ada, I specified that the library manager had to be <u>easy to master</u>, yet able to <u>properly</u> manage the Ada program parts.

Q: What do you mean, "manage the program parts"?
A: Well, as the compiler completes its job, it sends compiled units to a program library. The library manager should, among other things, allow the compiler to automatically update units and to mark (or automatically recompile) units that are outdated. In addition, the user should be able to delete unnecessary units and to display the library contents in some "user-sympathetic" way.

Q: OK, what about the other tools?
A: Well, notice that the MATSE model includes a "specification viewer". The Ada language allows for the creation of "specifications" that can be separately compiled into a library. The information included in these specifications is available for use but is often "physically separated" so the programmer cannot see (textually) the information. The specification viewer allows the programmer to temporarily display (perhaps using a split screen) the contents of any Ada specification.

Q: That sounds convenient.
A: It is ... and, again, it ACTIVELY supports teaching the Ada language.

Q: I notice you have also included file transfer capability in the MATSE.
A: Yes. I included this to take advantage of the teaching utility of the multiuser, mini-computer in our department. The workstations in our micro-lab are networked to this minicomputer. The file transfer capability allows the user - directly from the editor - to pass files back and forth between the workstations and the minicomputer. The student can then, for example, build an Ada program on the workstation and then transfer it to the minicomputer system

for compilation and for use by other students in a group project. This is also a great tool for demonstrating the portability of Ada.

Q: What about this thing you call a "pretty printer"?
A: Again, in an attempt to ACTIVELY teach Ada in a software engineering context, I wanted to provide tools that would automate tedious programming processes so that the student could concentrate on a more abstract view of software design and programming. The pretty printer simply formats our code in a consistent, readable way.

Q: The last tool I notice is the Ada Computer-Assisted Instruction tool.
A: Right. It seems to me that we can ACTIVELY enhance and personalize Ada education by providing "on-line" Ada instruction. I'm not sure that such a tool has to be directly available as an integrated tool from the editor though that is a possibility. In the design of the MATSE, I was looking for a CAI system that employed plenty of examples and was easy to use, logically organized (preferably similar to the Ada Language Reference Manual), and able to "mark" progress of a user in terms of lessons completed.

Q: And that collection of tools is what you call the Micro-Ada Training Support Environment or MATSE?
A: That's it!

Q: Sounds like a great concept. Any luck in building the MATSE?
A: Well, believe it or not, I had great luck. It turns out that several environments are "on the street" and have many of the tools specified in the MATSE model. I eventually chose a package called the "Ada Workstation Environment" (AWE) marketed by AETECH corporation as the foundation for the MATSE. This environment is centered around a full-screen, template-driven editor. On entry to the environment, the user is prompted for the file to be edited and the name of the programmer. Once inside the editor, the user invokes tools by simple, logical key strokes. For example, one key produces a file "prologue" or header with the programmer's name, file name and date automatically entered. Another key invokes a menu of possible Ada <u>type</u> templates and another invokes a menu of Ada <u>unit</u> templates. (Selection of a template causes that structure to be presented - syntactically correct - at the cursor position.) Other keys offer direct access to the file manager, the file transfer utility, the operating system and the library manager. "Help" is always available from a bar menu that "pops-up" at the bottom of the screen. It's a very "comfortable" environment.

Q: Does this environment come with its own Ada compiler?
A: No, it simply sits as a shell on "top" of a compiler. I chose to use the AdaVantage compiler produced by Meridian Software Systems.

Q: Does the environment that you use at West Point have all of the tools you describe in your MATSE model?
A: I was able to employ all of the tools except the syntax checker and pretty printer. It turns out that those two tools became available soon after I committed to the AWE and Meridian compiler. The library management tools, accessible through the environment, came with the Meridian AdaVantage package.

Q: What about the Computer Assisted Instruction tool?
A: I chose a CAI package, called "AlsyED - Lessons on Ada" produced by ALSYS, Inc.

Q: Are you satisfied with the MATSE as it is currently built.
A: Yes, very much so.

Q: How many workstations with the MATSE do you have.
A: We have 18 workstations. Each of the machines has a hard disk drive and 640 kilobytes of internal memory. Each workstation also has access to a printer.

Q: Do each of your students also have the MATSE on their personal computer?
A: No ... for several reasons. First, the stock "iron" that each student receives is a Z-248 with two floppy disk drives and 512 kilobytes of RAM. The MATSE requires a hard disk drive and 640K. Secondly, we have only been issuing personal computers at West Point for 2 years. Therefore, a large number of the upper two classes do not have personal computers.

Q: I'll bet you would like to have a MATSE for each student!
A: Oh, absolutely! As an interim measure however, I'd like to get a "MATSE simulator" for each student. That is, each student should have a MATSE "shell" (editor, file transfer utility and syntax checker) on a single floppy disk. A student could conveniently use the simulator ... with a user interface identical to the actual MATSE ... on his or her own computer for initial program editing and syntax checking.

Q: Obviously, this approach is affordable as well.
A: Right. Another idea that I've been working with is the possibility of adding small, primitive robots to each system (or group of systems). We have such a robot that we use for low-level research at West Point ... complete with stepper motors, lights, ground trackers, bumpers, a speaker and an

optical scanner. Instructions for controlling the robot can be embedded in high-level language commands and then transmitted, via a serial link, from a host machine to the robot's on-board processor. Thus we are able to write Ada programs to control the simple robot.

Q: Sounds fun!
A: It is fun! And that is exactly why I'm interested in adding such a "tool" to the training system. To actively teach Ada, I think we should try to make it fun!

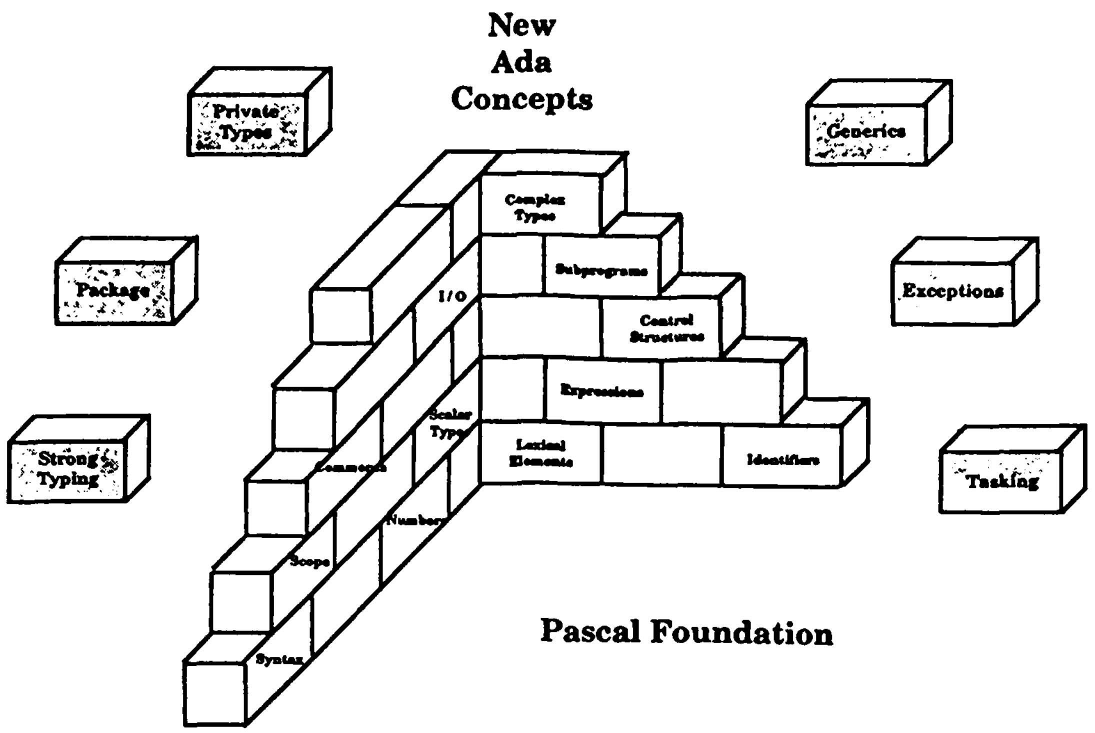
Q: Did you ever use the robot in your course?
A: We sure did ... it turned out to be a great success. The robot is a perfect subject for the object-oriented design method.

Q: So the MATSE is your cure for the "Turbonic Plague"?
A: Well, its part of the cure. There is actually one more "target".

Q: Oh yes ... the third "target".
A: As I indicated before, I've incorporated the MATSE into a teaching philosophy that I call "ACTIVE Ada".

Q: Tell me about it.
A: OK. My experience in teaching Ada over the last three years leaves me with the feeling that we are too "passive" in our approach. We often teach the course as a collection of "lessons in Ada syntax" requiring students to write 3 or 4 Ada programs. Such a course is a waste of time. We need to teach people to "think in Ada".

Q: Go ahead...you're getting excited aren't you?
A: I sure am! I decided to be much more aggressive in my approach to teaching Ada. First, as the next graphic illustrates, I decided to take advantage of the fact that our Ada students have all completed several courses using the Pascal language. Since Pascal is the "foundation" of Ada, I felt we needed to devote only a small amount of time replacing bricks (the "complex types" brick, for example) and refreshing the mortar of that foundation before we began to build upon it!



Building Ada Programmers on the Pascal Foundation
(Figure 3.)

Q: Interesting approach!
A: Secondly, I decided to teach Ada as it should be taught ... not as a collection of lessons in syntax but in a software engineering context. I decided to employ the text Software Engineering with Ada, by Grady Booch.

Q: Go on.
A: Next, I decided to teach my class in the micro-lab. 32 of the 40 lessons were designed to include hands-on Ada work using the MATSE.

Q: So students get daily exposure to Ada.
A: Exactly. For example, in Lesson 21 of "Software Engineering with Ada", we discuss "Generics". I introduce the topic by an example, briefly discuss rationale and syntax followed by a detailed study of the semantics and use of generic units. That discussion is followed by a short exercise in which each student builds and instantiates a simple generic unit. These daily sessions are aimed at improving skills with particular features of the language. I find that we could then devote more time to software design and engineering when working on major, out-of-class programming projects!

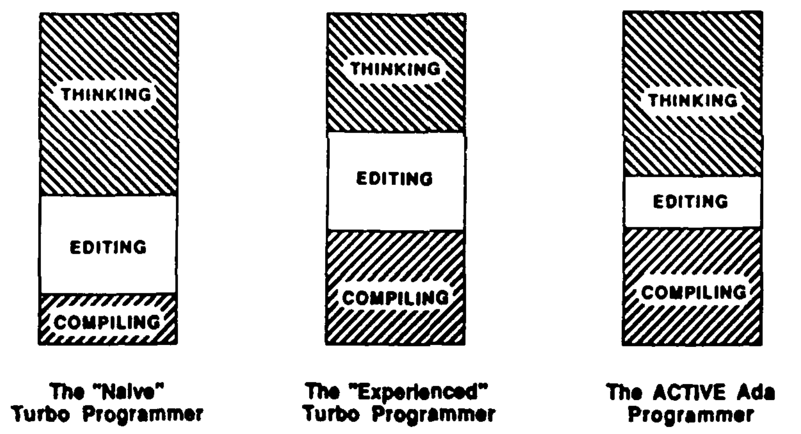
Q: You're on a roll ... keep going!
A: Lastly, I made aggressive use of the Ada CAI package. It turned out to be a great way to augment my instruction ...

especially in areas of Ada, like generics and tasking, that were new to the students.

Q: So that's "ACTIVE Ada" ... your cure for the "Turbonic Plague"?
A: That's it.

Q: Will it provide the cure?
A: Well, I'm hesitant to make an outright claim that I've found the cure. But, I can honestly say that I'm very satisfied with the progress that we made in one academic semester.

Q: What are some of the signs of progress?
A: Let me offer a modified version of the bar chart. This time I've added a third bar ... which represents the "ACTIVE Ada programmer". You'll note that the amount of time allocated for "up-front thinking" has increased.



The Curing Process
(Figure 4.)

Q: Interesting. Is this another "gut-feeling" chart?
A: It is ... but again I suspect that a lot of people agree with what it illustrates. I'm convinced of three things. First, the use of Ada is itself an effective cure. Through its powerful and integrated structures, Ada actively facilitates the "thinking" (software design) process. Secondly, by presenting Ada in a software engineering

context, methodical approaches to building software - especially large, complex software - are emphasized to the degree that they are used because they make sense. And lastly, the MATSE and the ACTIVE Ada philosophy actually make Ada personal, easy-to-use and FUN!

Q: Great! Anything else?
A: Well, I guess just a little advice and a simple request. My advice ... beware of "rat-like" behavior! And my request ... please do your part in "Curing the Turbonic Plague"!

Teaching Ada to Undergraduate Students

by

Jagdish C. Agrawal
Professor and Chairman
Computer Science Department
Embry Riddle Aeronautical University
Daytona Beach, FL 32014

Abstract: This paper is based on the experience of the author in course development and teaching Ada to undergraduate students. The common language standard across vendors is a definite advantage that should be capitalized on by the use of ANSI/MIL-STD 1815A as a required course material. However, Ada is merely a tool in the software development process, the larger emphasis should be on principles, processes, and goals of software engineering. Some exercises that allow the student to contrast Ada's capabilities with those of other languages should also be used.

1. Introduction.

Teaming up with Dr. Thomas Hilburn, this author experimented teaching Ada Programming Language as a sophomore level, special topics course in the Fall 1987 semester. Later, the author also experimented on including Ada as part of a junior level course on Software Engineering in the Spring 1988 semester. The author also observed (through literature searches, and recent conferences) what other educators were doing in this area. Based on the author's experiments, student feedback, and information collected from other universities, this paper presents a methodology for teaching Ada to undergraduate students with the goal of preparing these students for the job of Ada programmers in a large Software Engineering Project Team.

Let us first look at the current practice of preparing computer science majors in the undergraduate curricula of various universities. Based on the recommendations of the Association for Computing

Machinery (ACM) for a Computer Science Curriculum referred to as CURRICULUM 78, the core of an undergraduate computer science curriculum has included two programming courses -- CS1 and CS2, Computer Programming I and Computer Programming II. While implementation details of CS1 and CS2 may differ slightly from university to university, software life cycle approach in particular and software engineering concepts in general are largely missing from CS1 and CS2. This approach has been questioned by many educators, which can be summarized by the following quotation from a recent paper by Gibbs and Tucker [1]:

> "... core curriculum in computer science is frequently questioned because it seems to be composed of a collection of different programming and applications courses and fails to explicate adequately the principles that underlie the discipline."

The ACM now has a Task Force chaired by Dr. Peter J. Denning on the Core of Computer Science. The Tak Force recently issued a preliminary report [2]. The preliminary report recommends redesigning CS1 to include the following topics among others:

- software engineering
- data abstraction
- fundamental algorithmic concepts

Recently, a number of educators have advocated integration of Software Engineering into the entire first year programming classes [2-5]. However, there are several difficulties that are encountered by those who attempt such integration. Among these difficulties are shortage of text books that use such an approach, and limited teaching experience in this area. Yet the benefits far outweigh the difficulties. Providing fundamental concepts of software engineering before the student does any bad programming is more likely to prevent the formation of bad habits. Ada alone cannot guarantee high quality programs. Without a serious exposure to the principles, processes, and goals of software engineering, one could very well write bad COBOL or FORTRAN programs in Ada!

2. Introduction of Software Engineering Goals and Processes

Properties that are sufficiently general to be accepted as goals for the entire discipline of software engineering should be introduced. Useful reference material on this subject includes a book by Grady Booch [6], and a paper by Ross, Goodenough, and Irvine [7]. Students of the author used both of these and several others as reference material. Four properties that were introduced as the goals of software engineering were:

Modifiability: The software system should be suitable for controlled change in which some parts or aspects remain the same while others are altered, all in a way that a desired new result is obtained. This goal is called Modifiability [7].

Efficiency: The software system should operate using the set of available resources in an optimal manner. This goal is called Efficiency [7].

Reliability: The software system should not fail in conception, design, and construction, as well as it should have the ability to recover from failure in operation or performance. This goal is called Reliability [7].

Understandability: The system being developed should be an accurate model of the real world, providing a bridge between the particular problem space and the corresponding solution. This goal is called _Understandability_ [7].

While these properties have been cited in the literature [e.g., 6, and 7], the field of software engineering has not matured enough to produce precise mathematical measures for these goals. Therefore, any evaluation of how well the goals were achieved in a given software engineering project will be subjective to a large degree.

Using similar references, it is important to define and introduce the processes of software engineering and make references to these processes during different projects.

1. _Purpose_: Purpose crystalizes an objective. For example, the requirements of a system that is to be developed present a purpose.

2. _Concept_: Concept formulates how the purpose can be achieved. For example, the architecture of the software system to be developed to satisfy the requirements may be viewed as the concept.

3. _Mechanism_: Mechanism implements the structure of the concept.

4. _Notation_: The programming language for implementation of the system, which in our case was Ada, was going to be the notation for this course.

5. _Usage_: How the system is controlled. This allowed the instructor to introduce the relevance and importance of associated documentation.

Students were able to develop intuitive understanding of how the careful/careless choices for the processes of software engineering project can impact on the degree to which the goals of software engineering will be achieved in the project.

3. Introduction of Software Engineering Principles

Introduction of software engineering principles is a very important prerequisite to teaching Ada to undergraduate students. The greatest strength of Ada lies in its ability to implement the principles like abstraction, hiding, modularity, localization. This strength has been achieved by certain features of Ada that to a novice may appear substantially more complex than what other programming languages more traditionally used for CS1 and CS2 have! Unless the beginning student is introduced to the principles that Ada strongly supports, the student may entirely miss the appreciation of the beauty and strength of Ada! Ada, like any other programming language, is a tool for software development. Like any other software tool, this tool can be used successfully or misused, depending on its user. Any course that teaches Ada should also emphasie its capabilities to implement the principles of software engineering. Therefore, the following principles [6, 7] must be introduced:

Abstraction: Extracting essential properties while omitting inessential details. Ada provides a means to implement various levels of abstractions, both algorithmic as well as with objects.

Completeness: Ensuring that all of the important details are present.

Confirmability: Explicit statement of the information needed to verify correctness.

Hiding: Making inessential information inaccessible. Ada strongly supports the implementation of this principle.

Modularity: Purposefully structuring into relatively independent parts to easily achieve some purpose. Ada provides strong support for implementation of this principle by allowing the procedure-oriented modules as well as the object-oriented modules. A good designer can design modules that are relatively independent of the others, providing loosely coupled modules. Ada also allows the designer to control how tightly bound or related a module's internal elements are to one another, providing a means for strong cohesion.

Localization: Bringing related things together into physical proximity. This principles helps in creating modules with loose coupling and strong cohesion. Ada supports this principle.

Uniformity: Using consistent notation among all modules to directly support the goal of understandability.

4. Basic Ada Concepts

The first five contact hours of the course could be profitably devoted to the concepts discussed in the preceding sections followed by a written, closed book quiz on the concepts discussed. The next ten contact hours can now be devoted to some of the basic Ada concepts listed below. During this time, the students should be given some homework assignments to motivate them to absorb the material. The next two contact hours should be devoted to integrating all the knowledge imparted so far by using a set of "hands on" exercises of the type discussed in the next section. One large test should be administered and graded.

1. Four forms of program units in Ada: packages, subprograms, task units, and generic units -- Sections 6, 7, 9 and 12 of the Ada Language Reference Manual (LRM) [8].

2. Brief discussion of package structure (Section 7.1 of the LRM), Package Specification and Declaration (Section 7.2 of the LRM), and Package Bodies (Section 7.3 of the LRM).

3. Compilation units, "with" context clause, and order of compilation, program library, and elaboration of library units (Sections 10, 10.1 through 10.5 of the LRM)

4. Visibility rules and use clause (Sections 8, 8.1 through 8.4 of the LRM).

5. Generic unit as a template, generic formal objects, generic formal types, generic bodies, and generic instantions, and text_io as an example of generic units (Sections 12, 12.1 through 12.4, and Section 14.3.10 of the LRM)

6. Predefined data types in the package standard-- Boolean, integer, float, and character.

7. Other data structures -- arrays and strings, enumerated data

types, subtypes and derived types.

5. Instantion of some generic packages for Input/Output

Immediately after providing an overview of the basic Ada concepts discussed in the previous section, it is important to make the students experience how well Ada supports the software engineering principles of hiding, modularity, and abstraction. To do this, students can be given a copy of the section 14.3.10 of the LRM, Specification of the Package Text_IO [8]. It should be clear to the students that the implementation details of various components of text_io are neither necessary, nor available. The instructor's goal here is to make them realize that they could still "customize" various components of text_io without knowing the implementation details of the generic text_io. They are given some sample instantiations and then given several problems to do input/output using these instantiations. Below is a listing of the sample instantiations they can receive. In such exercises, students are experiencing different levels of abstraction, and developing experience with hiding, localization, and modularity.

The beginning students in Ada need to do some hands-on exercises. However, they need some support for doing some input/output for their exercises, before they can get started. Therefore, they should be encouraged to do some experimentation and also heavy reading from the LRM on the relevant sections. There is a great need for a user friendly User's Reference Manual (URM) for Ada.

```
-- INSTANTIATION # 1 to create package int_io
--
-- ====================================================================
--    REF     :        Program # 1,  mainhwk1
--                     This is the first of four packages for mainhwk1
-- ====================================================================
--
with text_io;
package int_io is new text_io.integer_io(integer);
--
-- END OF PACKAGE int_io


-- INSTANTIATION # 2 to create package bool_io
--
-- ====================================================================
--    REF     :        Program # 1,  mainhwk1
--                     This is the second of four packages for mainhwk1
-- ====================================================================
--
with text_io;
package bool_io is new text_io.enumeration_io(boolean);
--
-- END OF PACKAGE bool_io

-- INSTANTIATION # 3 to create package flt_io
--
-- ====================================================================
--    REF     :        Program # 1,  mainhwk1
--                     This is the third of four packages for mainhwk1
-- ====================================================================
--
with text_io;
package flt_io is new text_io.float_io(float);
--
-- END OF PACKAGE flt_io


-- SPECIFICATION OF THE FOURTH PACKAGE pkghwk1 to do I/O of strings
--
-- ====================================================================
--    REF     :        Program # 1,  mainhwk1
--                     This is the specification of the fourth package
--                     (pkghwk1) of four packages for mainhwk1
-- ====================================================================
--
with text_io; use text_io;
package pkghwk1 is
procedure get_string (a : in out string);
end pkghwk1;
--
-- END OF THE PACKAGE SPECIFICATION OF pkghwk1
-- BODY OF THE FOURTH PACKAGE pkghwk1 to do I/O of strings
--
-- ==================================================================
--   REF    :       Program # 1,  mainhwk1
--                  This is the package body of the fourth package
--                  (pkghwk1) of four packages for mainhwk1
-- ==================================================================
--
with text_io; use text_io;
package body pkghwk1 is
-- ==================================================================
--  We use pkghwk1 to define the "get_string" procedure.  We use the
--  get_line (= new_line plus get) from the text_io supplied with
--  Ada.  We construct the data object called dummy which is a string
--  containing 100 characters.  We initialize dummy by storing a
--  blank space for each of those 100 characters.  However, when we
--  get the input from the default file (keyboard) using the get_line
--  procedure of text_io, dummy accepts a string of n characters,
--  where n is a natural number and n < or = 100.  Then we copy the
--  string stored in dummy into a, the in out parameter of this
--  procedure.
-- ===================================================================
--
procedure get_string (a : in out string);
  dummy : string (1..100) := (1..100 => ' ');
  n : natural;
begin
  get_line (dummy, n);
  a := dummy (a'first..a'last);
end get_string;
end pkghwk1;
--
-- END OF THE BODY OF THE PACKAGE pkghwk1
```

Students should be encouraged to do different implementations of pkghwk1. Students will learn from this experimentation that the implementation of this package is of no concern to the user. User has the specification of pkghwk1, and that is sufficient. The students should also be encouraged to run their exercise programs using pkghwk1 with different implementations of pkghwk1. At this point they will see that one can go inside the body of pkghwk1 and change its implementation, but the interfaces provided in the specification of pkghwk1 remain unchanged, and their exercise programs that worked with the previous implementation of pkghwk1 will continue to work with the new one. This will illustrate to them the value of modifiability in maintenance of large systems.

```
-- ===============================================================
--   REF    :        Program # 1,  mainhwk1
--                   This is mainhwk1
-- ===============================================================

-- Four packages were compiled for this program -- (1) int_io,
-- (2) bool_io, (3) flt_io, and (3) pkghwk1.  The first three are
-- instantiationsof the genericpackagesinteger_io, enumeration_io,
-- and float_io.  The third package called pkghwk1 allows the user
-- to read in a string of less than or equal to 100 characters from the
-- default input file (terminal) by calling get_string(a).
-- ================================================================
with text_io; use text_io;
with  int_io; use  int_io;
with bool_io; use bool_io;
with  flt_io; use  flt_io;
with pkghwk1; use pkghwk1;

-- ===============================================================
--   Students are given different exercises (requirements) for
--   input/output and manipulation of data of the types specified
--   in the three packages above.  They are asked to design and
--   write the procedure mainhwk1 to meet these requirements.
-- ===============================================================
```

6. Operators and Control Structure

When the students have learned some of thepredefined data types and their input/output, as well as some other data structures, they are ready for expressions and control structures -- basic loop; while loop; for loop; if ... then ... else constructs; case statements; and classical goto's. This is not much different than it would be in any other programming language. Sections 5.3, 5.4, 5.5, and 5.9 of the LRM and examples from any textbook can be used to introduce these topics. Students with prior experience of programming in other languages are likely to pick up this material very fast. Since the students have already developed experience with input/output of certain data types, they can be given exercises for "hands on" experience. A written test on this module is also good. The length of this module can be four contact hours including discussion of several examples.

7. Advanced Types and Subprograms

About half of the course is over by now and a mid-term exam will reveal the success rate so far. During the next quarter of the course, students are ready for learning more advanced types to implement the software engineering needs of hiding and abstraction -- private and limited private types. More interesting data structures through access types and record types can also be introduced at this point. Also, the students are ready for introduction to functions, procedures, operators, and the concept of overloading. The concept of exceptions and exception handler as one of the tools to assist in fault tolerance in the program can be introduced at this point.

8. Abstraction, Hiding, Modularity and Localization in Design

Students have been exposed to many capabilities of Ada during about three quarters of the course. The last quarter of the course they need to concentrate more on practicing some of the software engineering principles. Although the book _Software Engineering with Ada_ [6] was probably not written as a textbook for undergraduate students, I like the emphasis on a design approach. He takes a reasonable size problem, describes the requirements in plain English, shows how one would identify the objects and operations and decide on visibility. By using the principles of abstraction and loacalization, he shows the reader how one can develop a modular architecture of the system and establish interfaces for the modules. The time available for this module of the course has allowed me to do two design problems from Booch [6]. Class was divided into several teams and each team was assigned the task of designing a component of the system resulting from some minor changes I made to the requirements of the two design problems discussed in the class. Only interface specifications were available to all teams, no team knew the implementation details of what the other teams designed and developed. The integration team was supposed to make the whole system work together. It was a good experience for all. More books are needed that provide fully worked out examples of design problems from requirements to the stage of implementation. At a recent Software Engineering Institute (SEI) Conference on Software Engineering Education held in Fairfax, Virginia, April 27-29, 1988, there was much interest expressed by many educators in similar case studies/worked out examples of systems from requirements to specification to design to implementation. SEI has invited educators to submit the case studies they have used for sharing with others through SEI publications. My software engineering students enjoyed the design approach, although initially, they found the enormous documentation and practicing the principles somewhat frustrating and unpalatable. The student feedback at the end of the course encouraged the professor to continue the design and documentation approach.

It is important to point out that in both experiments done by the author, team teaching Ada as a Special Topics course and teaching Ada as part of a Software Engineering course, the students had somewhat advanced standing and all of them had completed at least the equivalents of CS1 and CS2 courses of Curriculum '78 and most were reasonably fluent in at least one higher level programming language. The author has not experimented with teaching Ada to students with no prior experience in any programming language. However, this experimentation shows that the needs of the Department of Defense, Air Force, Army, Navy, and their contractors for training their programmers in Ada can be abundantly satisfied with this approach through a contract with a university that has demonstrated success in doing this.

9. Conclusions

While the correct use of syntax is necessary for the success of the student in a course that teaches them Ada, a good validated compiler, and use of LRM help the student become incrementally fluent in Ada syntax. The instructor's examples, exercises and testing should be to emphasize the correct use of this tool by emphasizing the Principles, Processes and Goals of software engineering. For example, to enforce abstraction, hiding, localization, modularity, and uniformity principles, students should do team projects such that team A develops one module of a software system, Team B develops another module of the same system, and so on. Across the teams only the interface specifications are known, the implementation details are hidden. It should be the responsibility of all teams collectively to make all components work together and produce the required results. Similar experiments by this author were successful. Many organizations that need to train their highly motivated programmers, can use this approach and train the programmers in using Ada.

10. References

1. Gibbs, N. E., and Tucker, A. B., "A Model Curriculum for a Liberal Arts Degree in Computer Science," Communications of ACM, Volume 29, Number 3, ACM, New York, NY, March 1986.

2. Werth, L. H., "Integrating Software Engineering Into an Intermediate Programming Class," ACM SIGCSE Bulletin, Volume 20, Number 1, ACM, New York, NY, February 1988.

3. Ford, G., "A Software Engineering Approach to First Year Computer Science Course," ACM SIGCSE Bulletin, Volume 14, Number 2, ACM, New York, NY, 1982.

4. Gillett, W., "The Anatomy of a Project Oriented Second Course for Computer Science Majors," ACM SIGCSE Bulletin, Volume 12, Number 3, ACM, New York, NY, 1980.

5. Koffman, E. B., Stemple, D., and Wardle, C. E., "Recommended Curriculum for CS2, 1984: A Report of the ACM Curriculum Committee Task Force for CS2," Communications of the ACM, Volume 28, Number 8, ACM, New York, NY, August 1985.

6. Booch, G., Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

7. Ross, D. T., Goodenough, J. B., and Irvine, C. A., "Software Engineering: Process, Principles, and Goals," IEEE Computer," May 1975.

8. U. S. Department of Defense, Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, American National Standards Institute, Inc., 1983.

9. Austing, R. H., Barnes, B. H., Bonnette, D. T., Engle, G. L., and Stokes, G., CURRICULUM '78, A report of the ACM Curriculum Committee on Computer Science, Association for Computing Machinery, New York, NY.

This Page Left Blank Intentionally

AN ADA-BASED SOFTWARE ENGINEERING PROJECT COURSE

Bruce W. Johnston
Associate Professor of Computer Science
Mathematics Department
University of Wisconsin-Stout
Menomonie, Wisconsin 54751

1 INTRODUCTION

The past decade has seen the evolution of a variety of undergraduate software engineering courses. Typically it is a project oriented course focusing on the software development lifecycle (Leventhal and Mynatt, 1987). Several other models have been proposed with the major variations in the amount of software engineering theory versus the extent of the software project (Tomayko, 1987).

The choice of programming language and its supporting software development environment also impacts directly on such a course. This is particularly true for modern languages, such as Ada, that were designed to support the principles of software engineering. Agrawal and Harriger (1985) have emphasized the need for an Ada-based software engineering course. Results of early experiments with such a course have been reported by Lapalme and Lamy (1986) and Tam and Erlinger (1987). Both groups were very satisfied with the use of Ada despite some technical difficulties with early versions of Ada compilers.

The objective of this paper is to describe an Ada-based software engineering course for students with no prior Ada experience. It is a one semester course using the textbooks of Sommerville (1985) and Booch (1987). The class size is limited to 20 students. The software development environment includes the Telesoft TeleGen2 Ada compiler (version 3.10) running under VMS on a VAX-11/780. Table I gives an outline of major topics covered in the course.

This course has been taught seven times over the past five years at the University of Wisconsin-Stout. The typical student enrolled in the course is a senior Applied Mathematics Major. These students have a strong background in Pascal provided by a two semester CS1-CS2 sequence along with a one semester Data Structures course. Some of the students have had an industrial programming internship.

Table I Software Engineering Course Information and Outline

Days	Topic
3	Introduction to software engineering and the software lifecycle
7	Ada for Pascal programmers
2.5	Project overview
1.5	Assignment #1 presentations
3	Software requirements analysis
2	PROJECT PRESENTATION: system requirements review
6	Top-down structured design
1	Midterm exam
1	PROJECT PRESENTATION: initial design review
3	Object-oriented design using Ada (including Ada package taxonomy)
1	PROJECT PRESENTATION: object-oriented design review
3	Software implementation -- programming languages (including Ada exception handling)
1	PROJECT PRESENTATION: stub version demonstration
4	Software implementation -- programming methodologies (including configuration control, user interfaces, and Ada programming style)
1	PROJECT PRESENTATION: system demonstration #1
2	Software testing
1	PROJECT PRESENTATION: system demonstration #2
1	Software documentation and standards
1	Management of programmers and programming teams
1	Final written exam
1	PROJECT PRESENTATION: final system demonstration

The primary course objectives include the following:

- understand the software engineering goals of producing understandable, reliable, maintainable, and efficient software;
- understand the principles of software engineering including abstraction, information hiding, modularity, localization, uniformity, completeness, and confirmability;
- learn the methods and techniques of software engineering that express the principles listed above;
- participate in a large scale software development project that covers the entire software lifecycle.

Learning the Ada language is not considered a primary objective. But Ada is used as a mechanism for illustrating and implementing the principles of software engineering.

The course has the following three major organizational threads:

- software engineering theory;
- software developement with Ada;
- group software development project.

The material from these three areas are heavily intertwined during the semester. For discussion purposes, however, it is convenient to consider them individually.

2 SOFTWARE ENGINEERING THEORY

Material for theoretical aspects of the course is drawn from two textbooks and selected references. Two texts were selected to provide a balanced view of software engineering. Booch (1987) is the best book available on Ada taught from the software design viewpoint. It can easily be considered the definitive source on object oriented design using Ada. Booch (1987) also does an excellent job of describing and illustrating the critically important Ada packaging concepts.

The Sommerville (1985) text complements the Booch text by providing more complete coverage of methods for the entire software lifecycle. It addresses the topics of requirements analysis, top-down structured design, design validation, testing, documentation, user interfaces and software management that are not included in Booch (1987). Supplementary material is derived from the following references:

- o Bray and Pokrass (1985) -- Ada types and objects, configuration control;
- o Saib (1985) -- Ada exception handling;
- o Dolan (1984) -- top-down structured design;
- o Wiener and Sincovec (1984) -- software design;
- o Myers (1979) -- software testing;
- o Stahl (1986) -- user interface design;
- o Shumate and Nielsen (1988) -- Ada package taxonomy;
- o Gardner (1983) -- Ada programming style.

3 SOFTWARE DEVELOPMENT WITH ADA

The discussion of Ada begins with a set of graduated Ada example programs that review elementary features of Ada on the same conceptual level as Pascal. The intent here is not to teach a "Pascal-subset" of Ada, but rather to start with semantically familiar concepts and then move directly into the more important features of Ada such as packages, generics, exception handling, etc. Shumate (1984) is a good reference for these introductory examples.

The following individual Ada programs are assigned early in the semester to introduce each student to some of the important concepts about using Ada:

1. Simple example of the students own choosing (50-100 lines);

2. Draw a simple picture on a color graphics terminal using an instructor provided graphics package;

3. Package to support an abstract data type such as polynomials or rational numbers. Package includes routines to GET, PUT, add, multiply, etc. The instructor may provide the package specification.

The first assignment is a good technique for getting students started with Ada. Since each program is different, it stimulates dicussion of various problems that can arise. Each student gives a short oral presentation to describe their program. This gives the students an opportunity to practice talking software in front of a technical audience and in the process, previews forthcoming lectures on code reviews.

The second assignment is designed to give the student additional experience in using a library package. This assignment is also useful because most of the group software projects include an extensive amount of color graphics.

The third assignment provides a more complete package assignment including the topics of separate compilation, private types, operator overloading and procedure overloading. It is a good example of how substantial changes to the package body can be made without changing the package specification. Finally, it introduces some key ideas on the order of compilation and recompilation which are fundamental to large scale software development.

Time constraints prevent detailed discussion of several Ada topics that are important features of the language such as tasking and representation specification.

4 GROUP SOFTWARE DEVELOPMENT PROJECT

The most challenging aspect of teaching the course is selecting an interesting, real-world application of appropriate size and complexity. The projects used thus far are described in Table II. Although all of the projects listed are very realistic and practical applications, those that were contracted by some external agency were received most enthusiastically by the students. Whenever possible, representatives of the client agency are encouraged to attend the in-class project reviews and demonstrations.

Each class is divided into 2-3 project teams which each separately implement the selected project. The instructor selects the composition and manager for teams of 4-10 students. Usually, team managers are selected from those students who have had an industrial programming internship. The team approach encourages productive competition between groups and stimulates discussion of design alternatives and the exact interpretation of software requirements.

Table II Group Software Development Projects

Semester	Project	Language
Spring 84	Database System for Cable TV Network (University Project)	Pascal
Spring 85	LANDSAT Satellite Image Analysis System	Modula-2
Spring 86	Doppler Weather Radar Display System	Ada
Fall 86	Doppler Weather Radar Display System (NASA project)	Ada
Spring 87	Doppler Weather Radar Analysis System (NASA project)	Ada
Fall 87	World Geographic Mapping System	Ada
Spring 88	RHI Doppler Radar Analysis System	Ada

The project scheduling is closely coupled with the theoretical aspects of the course. As the course progresses through the various stages of the software lifecycle, the student teams perform the corresponding tasks for the project. Table III shows the schedule of project milestones. Two design presentations are included to allow the students to experience both top-down structured design and object-oriented design methods. These design presentations are followed by a regular sequence of system demonstrations. The demonstrations serve as a progress check since each demonstration will have project specific functional requirements that all groups should meet.

5 CONCLUSIONS

Since many of our graduates enter software engineering careers, they approach the class with a high degree of enthusiasm. The course has been well recieved by those industries where our graduates have been placed. Heavy student demand for the course has resulted in some students being turned away from the course.

Students who have recently taken the course have expressed the following concerns and ideas in regard to the course:

- o slow speed of the Telesoft Ada Language System;
- o awkward and inefficient Telesoft Ada Library System;
- o desire to expand the course into a two semester sequence;
- o desire to have additional time to learn the Ada language.

To properly interpret these comments it is important to recognize that Ada is a large and complex language. The complexity results from the inclusion of extensive security features while preserving the expressiveness of the language (Bray and Pokrass, 1985). Its size reflects the language requirements for the broad application domain for Ada software.

Several anticipated changes will act to mitigate these concerns. As the technology for compilers and supporting tools advances, the overall quality and cost of the software development environment should improve dramatically. As these changes occur, it will become possible to introduce the Ada language in the courses prerequisite to the software engineering course. In addition to solving some of the problems listed above, this would provide additional time to study other advanced topics in software engineering.

One must also recognize that there are many benefits to be gained by using Ada. Important features including packages for encapsulation, generics for reusable components, and tasking for concurrent and real-time systems are described by Evans and Patterson (1985), Lapalme and Lamy (1986), Tam and Erlinger (1987), and Sammet (1986). So despite the additional complexities imposed by the use of Ada, the many advantages that Ada brings to the software engineering process make it well worth the effort.

The use of Ada in our software engineering course has proved to be a very natural and productive combination. Coupling Ada with a good Ada Programming Support Environment provides the correct tools for performing programming-in-the-large and programming-in-the-many. Viewed in this way, Ada is truly more than just another programming language and, as such, provides an excellent vehicle for software engineering. Overall we have been extremely pleased with the course and look forward to continued improvements in the quality of the Ada language systems to further enhance the course.

REFERENCES

Agrawal, Jagdish C., and Alka R. Harriger, 1985: "Undergraduate Courses Needed in Ada and Software Engineering", ACM SIGCSE Bulletin, February, 1985, pp. 266-281.

Booch, Grady, 1987: Software Engineering with Ada. Second Edition, Benjamin/Cummings.

Bray, Gary, and David Pokrass, 1985: Understanding Ada. John Wiley and sons.

Dolan, Kathleen, 1984: Business Computer Systems Design. Mitchell Publishing.

Evans, Howard, and Wayne Patterson, 1985: "Implementing Ada as the Primary Programming Language", ACM SIGCSE Bulletin, March 1985, pp. 255-260.

Gardner, Michael R., Nils Brubaker, Arthur Cohen, Carl Dahlke, Brian Goodhart, Tina Lewis, and Donald R. Ross, 1983: "Ada Programming Style", Intellimac, 7 pp.

Lapalme, Guy, Jean-Francois Lamy, 1986: "An Experiment in the use of Ada in a course in Software Engineering", ACM SIGCSE Bulletin, February 1986, pp. 124-126.

Leventhal, Laura M., and Barbee T. Mynatt, 1987: "Components of Typical Undergraduate Software Engineering Courses: Results from a Survey", IEEE Trans. Software Eng., vol. SE-13, November 1987, pp. 1193-1198.

Myers, Glenford J., 1979: The Art of Software Testing. John Wiley and Sons.

Saib, Sabina, 1985: Ada: an Introduction. Holt, Rinehart, and Winston.

Sammet, Jean E., 1986: "Why Ada is Not Just Another Programming Language", Commun. ACM 29, 8 (August, 1986), pp. 722-732.

Shumate, Ken, 1984: Understanding Ada. Harper and Row.

Shumate, Ken and Kjell Nielsen, 1988: "A Taxonomy of Ada Packages", Ada Letters, March/April 1988, pp. 55-76.

Sommerville, Ian, 1985: Software Engineering. Second Edition, Addison-Wesley.

Stahl, Bob: 1986: "UIMS - A Guide to Designing Friendly User/Computer Interfaces", Precision Visuals, Inc. and The Interface Design Group, 16 pp.

Tam, Wing C., and Michael A. Erlinger, 1987: "On the Teaching of Ada in an Undergraduate Computer Science Curriculm", ACM SIGCSE Bulletin, February 1987, pp. 58-61.

Tomayko, James E., 1987: "Teaching a Project-Intensive Introduction to Software Engineering", Special Report SEI-87-SR-1, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, March, 1987.

Wiener, Richard, and Richard Sincovec, 1984: Software Engineering with Modula-2 and Ada. John Wiley and Sons.

The Importance of Object-Based Knowledge Representation In Software Development

Presented at ASEET Symposium, 6-15-88

by

Cliff Layton, Director

The Rogers Institute for Software Engineering
Will Rogers and College Hill
Claremore, OK 74017
918-341-7510--286

The Rogers Institute for Software Engineering,

The RISE

Was Established in Summer 1987 to

Specialize in Ada, AI and Software Engineering

Research

Development

Technology Insertion

Education

AI Video Training
PC AI
WINTER 1987 / $4.95 U.S. $5.95 CANADA
Special Price
a voyage into
THE OBJECT-ORIENTED UNIVERSE

Object-Oriented Programming and Databases

by Jacob Stein

Introduction

We are in an object-based software age!

Our natural problem solving mode is object-based!

Objects more effectively capture meaning than numbers, functions, files or DBMS!

When we represent knowledge in terms of objects, we capture meaning effectively and encourage our natural problem solving mode!

This is especially significant as our problems become larger and/or more complex!

Purposes of Presentation

- Define Object
- Consider the Importance of Object–Based Knowledge Rep.
  - In Problem Solving, In General
  - In Software Engineering
  - In (AI) Expert System Building
  - In DBMS

I. Object Definition

A. Encapsulation: Completeness Within a Well-Defined Boundary

B. Definite Interface: Defining and Allowing Interactivity (with Other Objects)

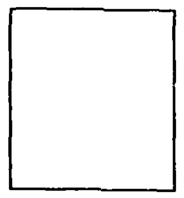
C. Loose-Coupling (Independence)

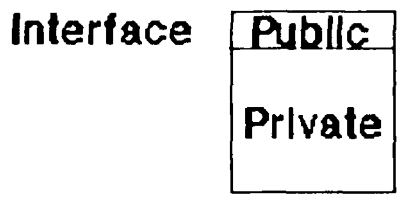
D. Inheritance: The Possibility of Gaining Definition from a Predecessor

E. Dynamic Binding

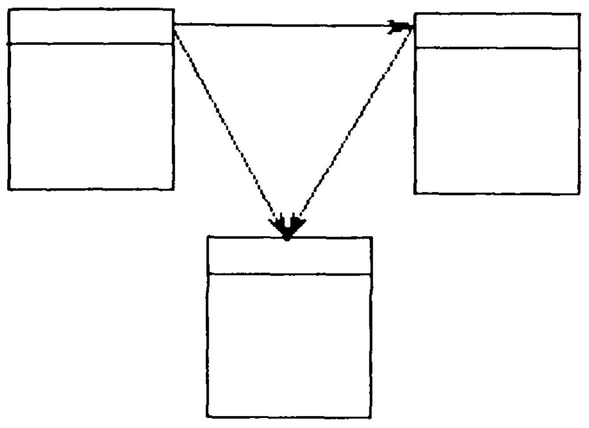
F. "Real World" Correspondence

G. Examples: A Ball, An Integrated Circuit, A Code Module (Ada)





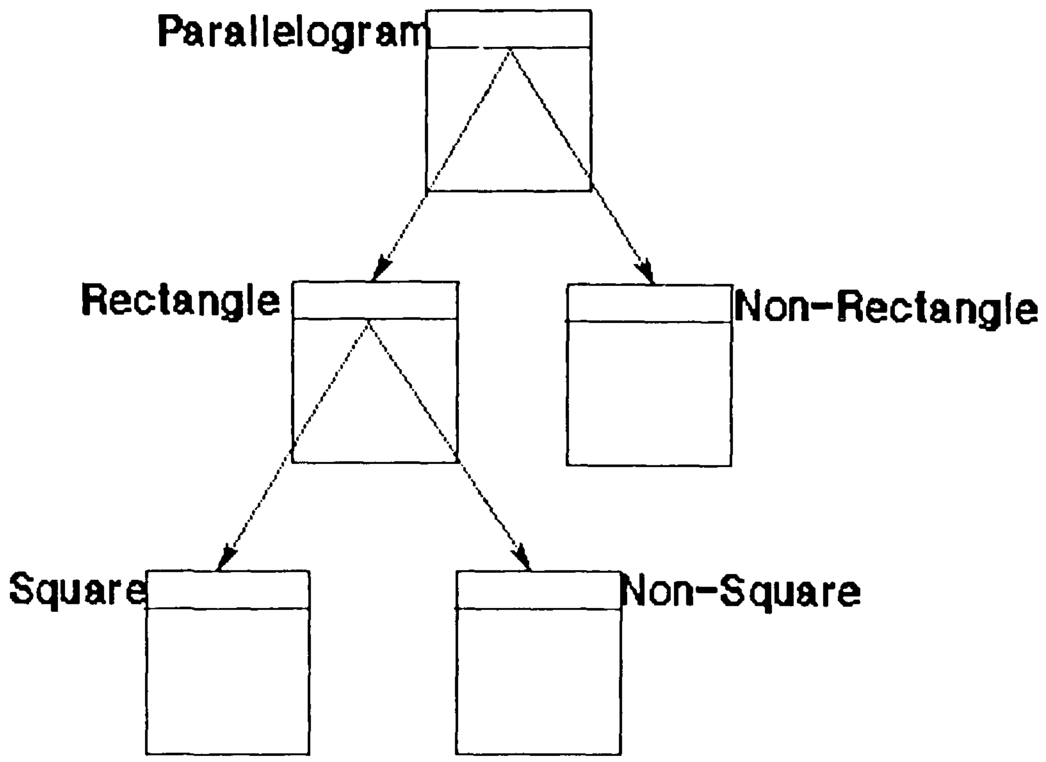
Encapsulation



Loose-Coupling

Lego Set

Dynamic Binding



Inheritance and "Real World" Corres.

II. Importance of Object-Based Knowledge Representation (KR) in Problem Solving

Problem description and solving using objects and manipulations is natural, pervasive and general!

Adequacy of Representation and Manipulation

"Real World" Correspondence

Distance Between Problem and Solution

A. KR and Manipulations Re. Actual Objects

B. KR and Manipulations Re. Drawings

C. KR and Manipulations Re. Natural Language

D. KR and Manipulations Re. Files and Databases

E. KR and Manipulations Re. Numbers

F. Computer-related problem solving is mostly relative to D and E above: shallow object-based KR.
   1. Not Highly "Real World" Corres.
   2 Large Distance Between Problem and Solution

G. New and more effective computer-related problem solving possibilities arise when KR is done relative to A through E above: deep object-based KR.
   1. High "Real World" Correspondence
   2 Small Distance Between Problem and Solution

III. Importance of Object-Based Knowledge Representation (KR) in Software Engineering

Software Engineering (SE) Concepts

Abstraction, Information Hiding, Modularity, Locatability, Uniformity, Completeness, Confirmability, Modifiability, Efficiency, Reliability, Understandability, Reusability

Each object characteristic, except dynamic binding, supports all of the above.

Object Characteristic	Example SE Concept
Encapsulation	Modularity
Definite Interface	Modifiability
Loose-Coupling	Abstraction
Inheritance	Reusability
Dynamic Binding	Reusability
"Real World" Corres.	Understandability

Use of object-based KR in SE is supported by
- methodologies
- CASE and other tools (?).

IV. Importance of Object-Based Knowledge Representation (KR) is Expert System (ES) Building

Object-based KR and manipulation can effecitvely represent and store meaning and expertise.

All general purpose expert system building tools (ESBTs) stress KR in terms of frames (objects).

General purpose ESBTs aid ES for classificaition, control, design, planning and simulation (not just classification or diagnosis).

LISP and other languages used for ES building are increasingly object-based.

Use of object-based KR in software engineering, expert systems, DBMS, and other software concerns can optimize the integration of SE, ES, DBMS and other tools and their products, and increase systems production through synthesis.

V. Importance of Object-Based Knowledge Representation (KR) in DBMS (Assumes Object-Based SE, ES and Problem Solving are Highly Appropriate)

A. Advantages

1. Relief of Burden of Storage of Objects in Memory

2. Optimization of Integration of DBMS with SE, ES, and other systems.

3. Storage of Heterogeneous Semantic Collections in Well-Defined Chunks

B. Disadvantages (Associated with Most New Tech.)

1. Very Limited Availability

a. Innovative Software Engr., Cambridge, MA

b. Ontologic, Billerica, MA

c. Servio Logic Corp,, Beaverton, OR

2. Slow Response

Conclusion

Object-Based KR allows optimal representation, storage and utilization of knowledge for general problem solving, software engineering, expert systems, and DBMS purposes.

Use of object-based KR in software engineering, expert systems, DBMS, and other software concerns can optimize the integration of SE, ES, DBMS and other tools and their products, and increase systems production through synthesis.

FOR FURTHER INFORMATION

Babcock, Charles: "Object-oriented systems emerge", COMPUTERWORLD, February 22, 1988, p. 25 and 30.

Booch, Grady: Software Engineering with Ada, Second Edition, Benjamin/Cummings, Menlo Park, CA, 1987.

Booch, Grady: Software Components with Ada, Benjamin/ Cummings, Menlo Park, CA, 1987.

Brooks, Fred: "No Silver Bullett: Essence and Accidents of Software Engineering", IEEE Computer, April, 1987, p. 10-19.

Buhr, R.J.A.: System Design with Ada, Prentice-Hall, Englewoods Cliffs, NJ, 1984.

Cox, Brad: Object Oriented Programming, Addison Wesley, Reading, MA, 1986.

Cox, Brad: "Building malleable systems from software 'chips'", COMPUTERWORLD (IN DEPTH), March 30, 1987, p. 59-68.

EVB Software Engineering Inc.: An Object Oriented Design Handbook for Ada Software, Fredrick, MD, 1985.

Gill, Philip: "MIS slowly warms up to object-oriented programming", COMPUTERWORLD (IN DEPTH), February 22, 1988.

Gosmi, Jeff and Desanti, Mike: "BOOPS and the Relational Data Base", AI EXPERT, November 1987, p. 60-66.

Layton, Clifford; Narotam, Matt; and Saib, Sabina: "OBJECT-ORIENTED DEVELOPMENT OF TRAINING SYSTEMS USING Ada", The National Security and Industrial Association, 1986.

Shafer, Dan: "THE OBJECT-ORIENTED Programming Universe", PC AI, Winter 1987, p. 29-32.

Stein, Jacob: "Object-Oriented Programming and Databases", Dr. Dobb's Journal, March 1988, p. 18-34.

This Page Left Blank Intentionally

The Software Engineering Institute An Overview

Gary A. Ford, Harvey K. Hallman
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract: The Software Engineering Institute is described, with special emphasis on its goals and activities with respect to education. Three projects are outlined, the Video Dissemination Project, the Graduate Curriculum Project, and the Undergraduate Software Engineering Education Project.

Introduction and Background

Because software is an increasingly critical element in many defense systems, the Department of Defense (DoD) is concerned with the timely production of quality software systems and components. Unfortunately, several constraints prevent developers from delivering high-quality software on schedule and within budget. If current trends continue, the cost to produce and maintain software could become prohibitive.

Recognizing the dilemma of the tremendous demands for quality software and the difficulty of producing it, the DoD established the Software Engineering Institute (SEI) and chartered to advance the state of software engineering practice. The SEI has a four-part mission:

- Bring the ablest professional minds and the most effective technology to bear on the rapid improvement of the quality of operational software in mission-critical computer resource (MCCR) systems.
- Accelerate the reduction to practice of modern software engineering techniques and methods.
- Promulgate the use of modern techniques and methods throughout the MCCR community.
- Establish standards of excellence for software engineering practice.

To accomplish this mission, a major objective of the SEI is to promote the evolution of software engineering from an ad hoc, labor-intensive activity to a managed, technology-supported discipline. The SEI strategy is to undertake a range of activities to identify critical software engineering problems, identify and develop solutions to those problems, and cause those solutions to be adopted.

SEI Programs

The SEI has established five *programs*, each with the responsibility to identify and develop solutions to problems within a critical area. Those programs are:

- Software Engineering Process Program
- Software Engineering Methods Program
- Software Systems Program
- Education Program
- Technology Transition Program

Each of these programs is summarized below.

Software Engineering Process Program. The software engineering process concerns the total set of software engineering activities needed to transform a user's requirements into software. Software engineering is the disciplined application of engineering, scientific, and mathematical principles, and methods and tools for timely and economical production of quality software. The objectives of the Software Engineering Process Program are to improve the maturity of software engineering as a practice, facilitate the effective introduction of available technology, and improve the quality and productivity of software development and maintenance.

Software Engineering Methods Program. The Software Engineering Methods Program strives to accelerate the development, introduction, and reduction to practice of methods, tools, and environments that improve software productivity and enhance quality. Specific objectives of the program are to identify, analyze, and propose solutions to problems that hinder the selection and application of technologies that promote modern software engineering techniques; to provide a common taxonomy for classifying and assessing methods, tools, and environments that facilitate software development; and to identify and recommend solutions for the improvement, standardization, and integration of this technology where such efforts reduce the risk of technology adoption and advance the state of software engineering practice.

Software Systems Program. The overall objective of the Software Systems Program is to pursue significant improvement in the development of real-time distributed systems. In order to do so, the program is building expertise and experience in selected application domains. The program also promotes the integration of software engineering with systems engineering, ensuring that software issues are addressed much earlier in system design; identifies and evaluates new software systems technology and software components that apply across application domains; and assists the real-time systems community in reducing the risk of adopting promising new technology.

Technology Transition Program. The Technology Transition Program is the focal point for SEI transition efforts. The program works with other SEI programs to identify needs, trends, and emerging technology in the DoD software community. The program also plays a key role in helping organizations prepare for the timely acceptance of current and emerging software engineering practices and technology. Its overall objective is to coordinate and manage SEI efforts to track the development of new practices and technology and to accelerate their adoption.

Education Program. The shortage of qualified software engineering professionals results from the increased demand for software throughout society and the extended period required for the educational system to develop new academic programs and to produce large numbers of graduates from those programs. The primary objective of the Education Program is to help increase the number of highly qualified software engineers by rapidly improving software engineering education throughout the academic, government, and industrial education communities.

The Education Program currently has four projects:

- Video Dissemination Project
- Graduate Curriculum Project
- Undergraduate Software Engineering Education Project
- Advanced Learning Technologies Project

The first three of these are described below; the fourth is the subject of a separate presentation at this symposium.

In addition, the Education Program sponsors the annual SEI Conference on Software Engineering Education, which was established to provide a forum for exchanging ideas and information on software engineering education and experiences with SEI-produced educational materials. The major objectives are to encourage educators to share their experiences and educational materials and to increase the visibility of the SEI in the software engineering education community. Participant reaction to the first conference indicates that it will attain the stature of the successful annual Technical Symposium on Computer Science Education, sponsored by the Special Interest Group on Computer Science Education (SIGCSE) of the Association for Computing Machinery (ACM).

Video Dissemination Project

The Video Dissemination Project (VDP) produces and delivers graduate level courses on software engineering methods to software practitioners in cooperation with the academic community. The courses are offered for credit by the local college or university. The courses are aimed at individual software developers in government and industry as well as continuing academic students.

To accomplish this, the SEI is establishing a Software Engineering Video Network of colleges and universities throughout the United States, with extensions into Canada and Europe. The industry and government software development groups that have a need for software engineering education for their people contact the SEI. The Video Dissemination Project works with them to locate a college or university close to their facility that will act as a focal point for the courses. The VDP provides tutor training where necessary and videotaped lectures of the courses and class materials. These tapes have been specifically recorded for this purpose during lectures being given to Carnegie Mellon and SEI students. After each class, the tapes are duplicated and sent, along with the classroom material, to the participating universities for use by their tutor. The local tutor participates in discussions of the materials with the students and grades the homework and exams. Successful students receive credit from the participating university. Tutors and students can call the teacher at the SEI to discuss the material with them. In the near future, the courses will also be available as live satellite broadcasts. All participants benefit from the standards established by Carnegie Mellon and the SEI for quality of education.

It is the objective of the VDP to increase the number of qualified teachers of software engineering. The *tutored video* format is used, not only to increase the students learning [Gibbons 77] but also to encourage the faculty at the participating institution to develop their own teaching expertise in the subject matter. Some departments have a shortage of qualified staff. Others have a shortage of funds needed to start a new curriculum. The VDP provides a valuable assistance to these institutions as they step up to providing industry with a solution to their needs.

A pilot course, "Formal Methods in Software Engineering", was taught in the spring semester of 1988 at eight participating colleges and universities in addition to Carnegie Mellon University: California State University at Sacramento, East Tennessee State University, Florida A & M University, George Mason University, Johns Hopkins University, Monmouth College, The University of North Carolina at Chapel Hill, and The Wichita State University. Participation is now open to any interested academic institutions.

The Education Program has developed a curriculum for a Master of Software Engineering degree. It identified the requirements for such a degree in 1986 when it sponsored a workshop at SEI to identify the problems in software engineering education [Gibbs 87]. It developed a specification for the such a degree in 1987 [Ford 87]. It held a workshop on the design of an MSE curriculum in March 1988 and will publish a report on "The Design of an MSE Curriculum" shortly [Ardis 88].

The proposed Master of Software Engineering has six core courses that are being developed for delivery on the Software Engineering Video Network: Software Project Management, Software System Engineering (including requirements analysis), Software Specification, Software Design, Software Verification and Validation, and Software Generation and Maintenance. Additional electives will be offered over the network as they are available. It is anticipated that the participating university will provide many of the electives necessary from their own computer science staff, in such subjects as software environments, networking, data structures or other areas, where they have expertise.

Some participating colleges and universities will stop using the materials developed by SEI after they have become confident that they can teach the courses themselves. Others will find it convenient to continue using the network as the source for some or all courses in their graduate program. Some participants will add "Software Engineering" options to their already existing Master of Computer Science degrees while others will begin offering the MSE degree. Most of these colleges and universities will have started meeting the demand for educating software engineers earlier than they would have if the Software Engineering Video Network had not existed.

Graduate Curriculum Project

The purpose of this project is to promote software engineering education at the graduate level. The project has two major goals:

- Identify, organize, and document the body of knowledge that might be taught in graduate-level software engineering programs.
- Design, develop, and support a curriculum for a Master of Software Engineering (MSE) degree.

Graduate-level education includes both university and continuing education programs in government and industry. The audience for software engineering education is large and diverse. These disparate groups need curricula with different durations, structures, and emphases. Therefore, the Education Program made a decision to define the content of software engineering education in curriculum *modules*. Each module presents a highly focused topic and normally contains less material than a typical university course. Each module is embodied in a document that includes a detailed, annotated outline of the material, an annotated bibliography, and other information of use to instructors. A variety of courses and degree programs can then be constructed from the modules.

The SEI was chartered partly in response to the need for a greatly increased number of highly skilled software engineers. The Education Program believes that this need can best be addressed by encouraging and helping academic institutions to offer a Master of Software Engineering degree; therefore, we have concentrated initial curriculum design efforts in support of this goal. Curriculum documents, however, are not sufficient to teach software engineering effectively. Well-written texts, exemplary software, sample documents and forms, case studies, exercises, and other support materials are also necessary.

Perhaps the most important support material an educator can have is a good textbook for students. The author of a textbook must have technical expertise, knowledge of education, writing skills, and a substantial amount of time to commit to the project. Because the royalties from a commercially successful book provide adequate rewards for this kind of effort, it is not necessary for the SEI to hire authors. Instead, the SEI serves as a catalyst for developing textbooks and monographs on software engineering for practitioners and students. To support this activity, the SEI has entered into an agreement with Addison-Wesley Publishing Company to publish the *SEI Series on Software Engineering*. The first volumes in the series are expected early in 1989.

In cooperation with Boston University, Arizona State University, the University of Maryland, and University of California, Irvine, the project is investigating the use of a large, delivered software system coded in Ada to determine its value as an educational vehicle. After acquiring such an artifact and the rights to redistribute it, these universities, which comprise the artifact User Committee, will report to the SEI on its effectiveness in teaching software engineering.

Two major uses of the artifact are to be considered. First, it can serve as an object of study in ways similar to those of the traditional engineering disciplines. For most students, it will be the first time they have seen a large system with its associated documentation. Just seeing a system of this size will help to motivate many of the principles and techniques of software engineering. Second, the artifact can be the basis of large-scale activities and experiments, including maintenance and enhancement, configuration management, and testing.

Undergraduate Software Engineering Education Project

In terms of the numbers of students affected, the Education Program can make its greatest impact by influencing undergraduate computer science curricula. The purpose of this project is to exert that influence in effective ways. The project has three major objectives:

- Promote an increased awareness and understanding among educators and their students of software engineering and its differences from computer science.
- Promote improved undergraduate computer science curricula and an increased capability among students completing a bachelor's degree in computer science to perform productively as software engineers.
- Develop and promote development of educational materials to support enhanced software engineering education in undergraduate computer science programs.

Project activities that will help achieve these objectives focus on dissemination of knowledge and materials to educators, who in turn can deliver that knowledge to their students. By making educators partners in this effort, the small project staff will be able to reach many students.

The project will employ several specific mechanisms to achieve its goals. Publications, conference presentations, and workshops will be used to promote a better understanding among educators of the scope and content of the software engineering discipline. Teaching materials will be distributed both in text form and in machine-readable form, as appropriate. Course designs and curriculum recommendations will be published in text form and publicized at conferences and workshops. Resident academic affiliates will help produce educational materials and will take an increased interest in and understanding of software engineering back to their own institutions.

During the fall semester of 1986, a senior-level software engineering course was offered at Carnegie Mellon University, taught by James Tomayko, a temporary member of the Education Program staff, on leave from The Wichita State University. The materials developed by him and his students were collected, edited, and published as *Teaching a Project-Intensive Introduction to*

Software Engineering (SEI-87-SR-1). Several universities have used these materials in support of their courses.

To investigate the suitability of Ada as an introductory programming language (including language and support issues), the project is sponsoring two pilot studies. In January 1987, hardware and Ada software were loaned to West Virginia University and The Wichita State University, both of which are academic affiliates of the SEI. These systems are being used in freshman-level courses, and the lessons learned are being documented and forwarded to the SEI.

Improving the software engineering content of introductory programming and data structures courses has been proposed as an important step in improving the overall undergraduate curriculum. To elucidate some of the problems of the freshman courses and to identify possible solutions, the project sponsored a workshop on Ada in Freshman Courses in June 1987. The participants included the authors of several successful programming textbooks, the chairman of two recent ACM curriculum committees that examined introductory courses in programming and data structures, and educators who had used or were planning to use Ada in freshman courses. The conclusions of this workshop have helped guide the development of the project plan (see *Report on the SEI Workshop on Ada in Freshmen Courses* (CMU/SEI-87-TR-44) for details on the workshop and its conclusions).

The project has also begun an effort involving the Ada Joint Program Office, the Carnegie Mellon University Computer Science Department, and two private companies to produce an Ada programming environment tailored especially for beginning programmers. The environment will not only include a compiler and program generation and debugging tools, but also Ada component libraries and library browsing tools, new course designs to use the environment, and additional educational support materials. This environment is the subject of another presentation at this symposium.

A second Ada-related effort is planned for 1989: the development of a handbook of information for educators considering adopting Ada as their primary language of instruction. Many educators remember the problems encountered when they changed from Fortran to Pascal in the 1970s, including finding appropriate educational resources (compilers, textbooks, exercises) and integrating the new language throughout the curriculum. The purpose of the handbook is to help educators anticipate the problems and provide suggestions for solutions. An example is the restructuring of some courses, especially the beginning courses, to take advantage of the significant new features of Ada. The handbook will also include information on Ada compilers for typical university hardware configurations and information on textbooks at various levels.

Affiliates Programs

The SEI has developed industry, government, and academic affiliate programs. These programs provide a mechanism for organizations to become formally affiliated with the SEI for the purposes of the exchange of information and cooperative projects. The Education Program administers the Academic Affiliates; the others are administered by the Director of Affiliate Relations.

To date, Academic Affiliates have made major contributions to the development of curriculum modules and educational support materials, have participated in four faculty development workshops, and have participated in the pilot course offering of the Video Dissemination Project. The lessons learned from affiliates have helped guide the SEI Education Program.

Summary

The Software Engineering Institute is undertaking a wide range of projects and activities to improve the practice of software engineering. A significant part of the strategy is to improve software engineering education. This can only be accomplished through cooperative efforts between the SEI and the members of the academic, industrial, and government education communities.

The Education Program strategy can be summarized as six kinds of activities:

- Identify, organize and document the rapidly evolving body of knowledge of software engineering
- Collect, develop, and distribute educational support materials of all kinds
- Design appropriate curricula for academic and industrial software engineering programs
- Implement and deliver software engineering curricula
- Promote the development of software engineering faculty in colleges and universities
- Bring advanced technologies to bear on the improvement of software engineering education

References

[Ardis 88] Ardis, M. A., "The Design of an MSE Curriculum", forthcoming SEI technical report, 1988.

[Ford 87] Ford, G., Gibbs, N., Tomayko, J., "Software Engineering Education, An Interim Report from the Software Engineering Institute", Technical Report CMU/SEI-87-TR-8, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA., 1987.

[Gibbons 77] Gibbons, J. F., Kincheloe, W. R., and Down, K. S., "Tutored Videotape Instruction: A New Use of Electronic Media in Education", *Science, Vol. 195*, March 18, 1977, pp. 1139-1146.

[Gibbs 87] Gibbs, N. E., Fairley, R. E. (editors), *Software Engineering Education, The Educational Needs of the Software Community*, Springer-Verlag, New York 1987.

This Page Left Blank Intentionally

Education & Training of Ada Lessons Learned

Edward Colbert
Abs(S/W)
Absolute Software Company
4593 Orchid Dr.
Los Angeles, CA 90043-3320
(213) 293-0783

Outline

Edward Colbert, Lecturer

Overview of Past Courses

Review of Selected Past Courses

Lessons Learned

Suggested Curriculums

Intro & Advance Class Outiines

How Small Companies Can
"Develop a Complete Training Program"

Conclusions

Edward Colbert
President - Abs (S/W)

Worked with Ada, Software Engineering in DoD Environment since '81

- Consultant at JPL on Real-Time Weather Processor (RWP)
- Consultant on Burger Test of Ada Proficiency, Psychometrics, Inc.
- Consultant at JPL on Global Decision Support System (GDSS)
- Principle Investigator on TRW's Ada PDL IR&D
- Security Engineering, Requirements Analysis, & Design on TRW's Army Secure Operating System (ASOS) Project
- Security Engineering on TRW's Inter-Service Agency Automated Message Processing Exchange (I-S/A AMPE) Project

Training Ada & Software Engineering since '82

JPL	Intel	TRW
SAIC	CONVEX	Litton
Cubic	USAF Space Division	Teledyne
UCLA	CSU-Fullerton	

Member of Ada Training Subcommittee of TRW's Electronics & Defense Sector's Ada Coordinating Group

Past-Chair of LA ACM SIGAda
Past-Chair of LA ACM SIGSoft

Overview of Past Courses

When	For Who	Times Offered	Duration hrs	ks	Class/ Week	People/ Class	Topics (hrs)	Compilers
'82-'85	TRW After Hours	7	36	12	1-3hr	20-25	Ada Programming	AdaED,ICC Verdix
Fall '84	CSU Fullerton	1	64	8	8-8hr	25-30	Ada Programming	R&R/PC
'85-'86	UCLA	4	36	12	1-3hr	20	Ada Programming	AdaED/PC TeleSoft 1.5/ Unix
'85	Teledyne Systems	2	8	1	2-4hr	20-25	Ada Overview	–
'85	Litton Data Systems	1	60	10	2-3hr	15-20	Ada Programming	–
'86-'87	SAIC	2 2 2 1	70 20 20 3	6 2 2 1	2-5.5hr 1	20-25 20	Ada Programming S/W Engineering & Design Ada PDL Manager's Overview of Ada	VAX/Ada
'87	Cubic Corp.	1	40	2	3-8hr	20	Ada as a Design Language	VAX/Ada

Overview of Past Courses (cont)

When	For Who	Times Offered	Duration hrs	Duration wks	Class/ Week	People/ Class	Topics (hrs)	Compilers
'87	TRW BMD	1	24	3	2-6hr	20-25	Intro. to Ada Program.	VAX/Ada
			54	9	2-3hr		Adv. Ada Program.	
			9	1.5	2-3hr		Object Oriented Design	
			12	2	2-3hr		Ada as a Design Lang	
			18	3	1-6hr		MSES Project Labs	
'87	CONVEX Computer Corp.	1	40	1	5-8hr	21	Advanced Ada Programming	Verdix/Sun
'88	Intel	1	40	1	5-8hr	8	Intro. to Ada Program.	
							Adv. Ada Program.	
'88	JPL/RWP Tech Staff	1	40	1	5-8hr	25	Intro. to Ada Program.	VAX/Ada
			40	1	5-8hr	25	Adv. Ada Program.	
			24	1	3-8hr	25	Object Oriented Devel.	
			16	1	2-8hr	25	Ada Run-Time Env.	
			16	1	2-8hr	25	Ada as a Design Lang.	
			40	1	5-8hr	25	Review/Special Topics	
	JPL/RWP Project Mgmt & Customer	1	40	1	5-8hr	25	Intro. to Ada Program.	VAX/Ada
			24	1	3-8hr	25	Adv. Ada Program.	
			16	1	2-8hr	25	Object Oriented Devel.	
'88	SAIC	1	55	8.5	2(3)-5hr	14	Ex. Intro. to Ada Prog.	VAX/Ada
			40	4	2-5hr	14	Adv. Ada Program.	
			25	2.5	2-5hr	14	Object Oriented Devel.	

Review of Selected Past Courses

SAIC Course Observations

Cubic Corp. Course Observations

TRW BMD Course Observations

CONVEX Computer Observations

INTEL Computer Observations

JPL/RWP Course Observations

Conclusions from Experiences

SAIC Course Observations

Lectures:

- Well Received
  - DSSD easy to teach as basic Requirements Analysis
  - Main advantage of DSSD is entity diagrams
- Students a little burned-out by 12th week

Labs:

- Use of TeleQuizes® used in 1st Class
  - Good support for "Simple Subset of Ada"
- Design Labs require tool support
- Walk-through of designs, good teaching technique
  - Must also have walk-through of "good" solution
- Management needs to require participation in labs
- Continuity of "Queue" lab exercises, good teaching technique

® TeleQuizes is a registered trademark of TeleSoft, Inc.

Cubic Course Observations

Lack of Continuity between Design & PDL course labs reduced effectiveness of PDL labs

- Spent more time on Requirements Analysis & Preliminary Design than studying PDL representation of design

- Current training program integrates courses more

High-level nature of Ada Course reduced students ability to effectively design.

- Spent lots of time discussing details of Ada

TRW-BMD Course Observations

Intro Ada Class started out **too fast** for “Engineers”
(FORTRAN Coders & Designers)

- Initial Preparation not done by most of class
- Directed to cover complete lecture on types during 2nd class (7 hours)
  - Too slow for “Computer Scientists”
  - Overwhelmed “Engineers”

Delay in starting class reduced usefulness

- Less “tailoring” than originally planned
- Design Classes too late for PDR
- Advance Class too late for helping in design

TRW-BMD Course Observations (cont)

Groups that did <u>all</u> of the labs are doing well!

Incremental labs	⇒	Reduced emphasis solving particular problem.
		Time for multiple problems
Multiple labs on Each Topic	⇒	Re-enforcement of concept
		Comparative application of concept

- Developed re-usable packages applicable to project

Project Specific Labs:

- Helped students understand application of Ada to project problems
- Made Ada real
- Showed that Ada can work!

CONVEX Computer Course Observations

1 of best class:

- Most Effective
- Best Students? (Most Motivated?)

Many students lost at start

- Students hadn't done initial reading & test
- 1 hour review stretched to 1 day (+ time throughout course)

Use of Incremental/Multiple labs effective

Availability of computer very helpful

- Each student had computer on desk.

Intel Computer Course Observations

1 of best class:

- Most Advanced Students
  - 1 had some experience with Ada
- Small Class

Covered Introduction & Advance Ada Programming Courses at 2X Usual Pace

- Students need exposure to “Intro.” Topics & Concepts
  - Didn't need to spend time explaining

No Labs

- Didn't really need Intro. Labs for re-enforcement
- Students did labs following week

Was not planned this way

JPL/RWP Class

Well Planned & Effective Training Program

Cover all Project Personnel

- Designers
- Programmers
- Testers
- Management
- Customer

Thorough Program

- Programming & Design Classes
- PDL Class
- Run-Time Environment
- Month-Long Project Case Study
- Follow-Up Classes

Labs

- Incremental/Multiple Problems
- Project Specific Problem

Weekly Feedback/Reviews

Testing

Lessons Learned Curriculum

Don't try to teach all Ada in 1 - 40 hour course

- Too much for student to learn effectively
- Superficial Coverage or No time for Lab
  - **Most** students need lab reinforcement

Don't attempt to teach S/W Engineering in Ada Course

- Teach S/W Engineering Principles & how Ada supports them

Lab work must be provided & **required**

- Tool support for Design & RA courses (& work) not required, but ....

Tailoring Classes to projects assists the learning process

- requires sufficient lead time to develop to obtain maximum benefits

Try to give separate classes for Advanced Personnel

Lessons Learned Curriculum

Provide a "complete" curriculum

- "Technical":

Requirement Specification	Requirement Analysis
Design Methodology(s)	Design Representation(s)
Ada Programming	Testing
Advance Seminars	Tool Support
Evolutionary Delivery	2167(A)
Security	etc.

- "Managerial":

S/W Development Management	Project Economics
Project Organizational Issues	etc.

- "Support":

Quality Assurance	Configuration Management
Maintenance	etc.

Lessons Learned Scheduling Training

Start classes sufficiently ahead of projects

- Costly to **Bid** project without knowledge of Ada & S/W Engineering
  - Incomplete/Unrealistic Understanding of Problem
  - Incomplete/Unrealistic Understanding of Cost/Schedule
- Costly to **Plan** project without Knowledge of Ada & S/W Engineering
  - Incomplete/Unrealistic Understanding of Problem
  - Incomplete/Unrealistic Understanding of Cost/Schedule
  - Insufficient knowledge to select tool support!
- Costly to **Start** project RA & Design without Knowledge of Ada & S/W Engineering
  - Lack Understanding Constraints
  - Lack Understanding of Potential Support
  - Must live with Incomplete/Unrealistic Cost/Schedule
  - Lack of tool support
  - Lack of Understanding how to Use tools/techniques
  - NO TIME

Lessons Learned
Is Ada Like Calculus?

Not everyone can learn calculus. It is the first rigorous exposure to formality and abstraction that we see in college. Facility with the principles of calculus is required to deal with many kinds of complex systems. We do not simplify calculus so more people can learn it.

.

.

.

Not everyone can learn Ada. Ada is the first language intended for wide-spread usage emphasizing formality and abstraction. It is wrong to expect that everyone who is presently doing software can learn Ada. Not everyone who learns algebra can understand calculus, not everyone who understands Bohr's atom can grasp Schroedinger's wave equations, and not everyone who understands FORTRAN or CMS2 can understand the abstractions of Ada.

Mark Gerhardt, TRW-ESL
"Don't Blame Ada", DS&E, Aug 1987

Is this correct?

Lessons Learned
Is Ada Like Calculus? (cont)

Not Exactly

Software Engineering is like Mathematics

- Ada is like a major subset of the formulas used in Calculus (Mathematics)

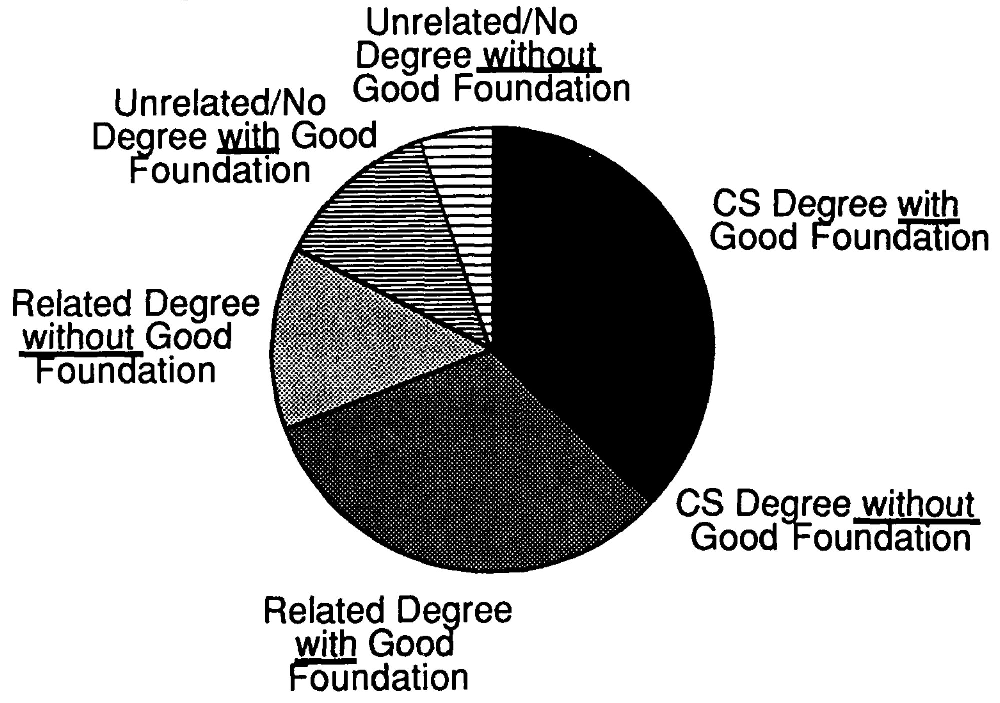
Ada/Software Engineering	**Mathematics**
Some people understand how to	**Some people understand how to**
program using only procedures, functions, etc.	solve algebraic problems
program with data abstraction in addition to above	solve differential equation problems
design software with data abstraction	apply differential equation to other fields (science, economics, etc).
implement pre-designed reusable program units	solve problems with integration
design reusable program units	apply integration to other fields (science, economics, etc).
Implement pre-designed concurrent software	solve problems with Partial Differential Equations
design concurrent software	apply Partial Differential Equations to other fields (science, economics, etc).

Lessons Learned

Some People Have Hard Time or Can't Learn to be "S/W Engineers"

Wrong Profession

- What is the background of the current population of "Programmers"



* Related Degrees include EE, Aeronautical, Physics, etc.

Fear or Lack of Motivation

- Often Related to above

Lessons Learned
How to Maximize Effective Population

Students should attend classes based on **Ability & Responsibilities**

- Content of Introduction Ada Course should be sufficient for most programming tasks
  - Separate Introductions for “Week Background” & “Strong Background”
- Provide “Remedial” Classes (e.g. Data Structuring)
- Advance Ada Courses should cover Special Features:Generics, Tasking, Low Level Features
- Teach about all phases of development
- Provide Advance Seminars (e.g. Designing for Reusability)

Lessons Learned
How to Maximize Effective Population (cont)

Provide & Require Continuing Education

- Make part of each job
- Provide time weekly, daily, monthly, ...

Give people positions based on **Ability**

- Don't let people who **fear** Ada &/or S/W Engineering Manage
- If you need their knowledge, put them in project consulting role

Identify Support Tools & Software (inc. reusable component libraries)

- Train personnel in their use

Lessons Learned
Testing

Touchy Subject:

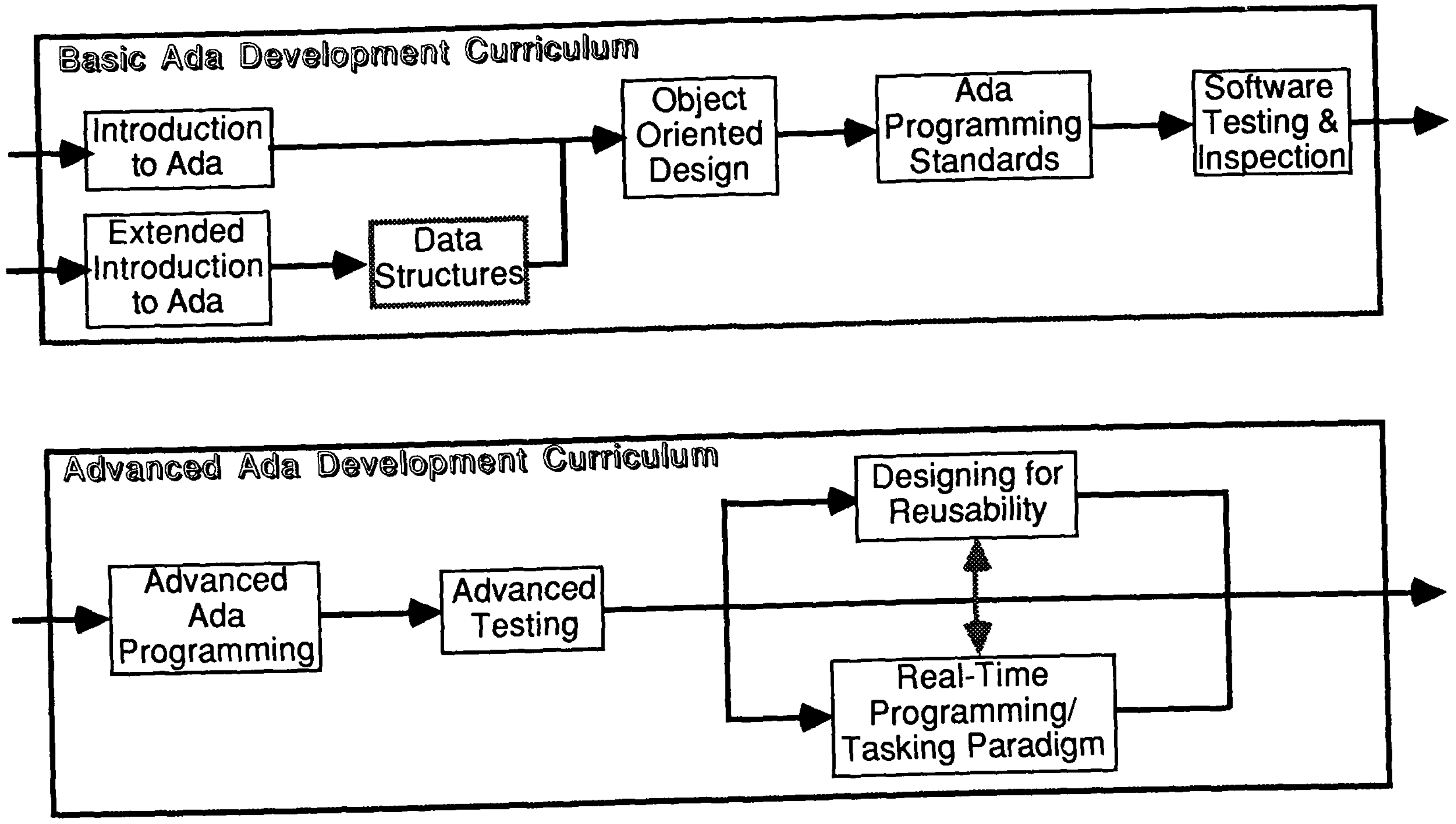
- **Employees fear it will affect career development, raises & promotions**
- Employers avoid it because:
  - Employee fears
  - What if Customers asked to see results(?)
- Controversy over Certification

Recommend Testing:

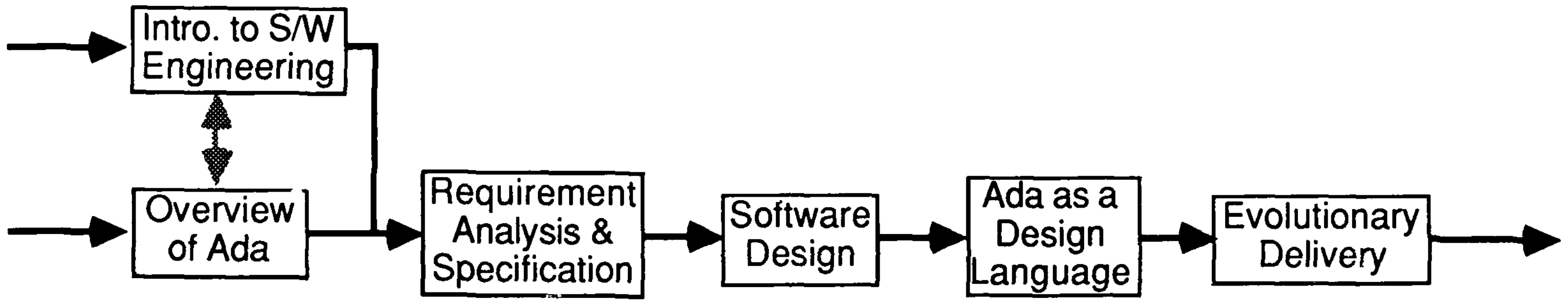
- Need information to effectively tailor students training program
- Need measure of students comprehension to determine Job Assignment
- Need feedback on effectiveness of training program

Must be prepared to provide additional training to students who wish to work on weaknesses

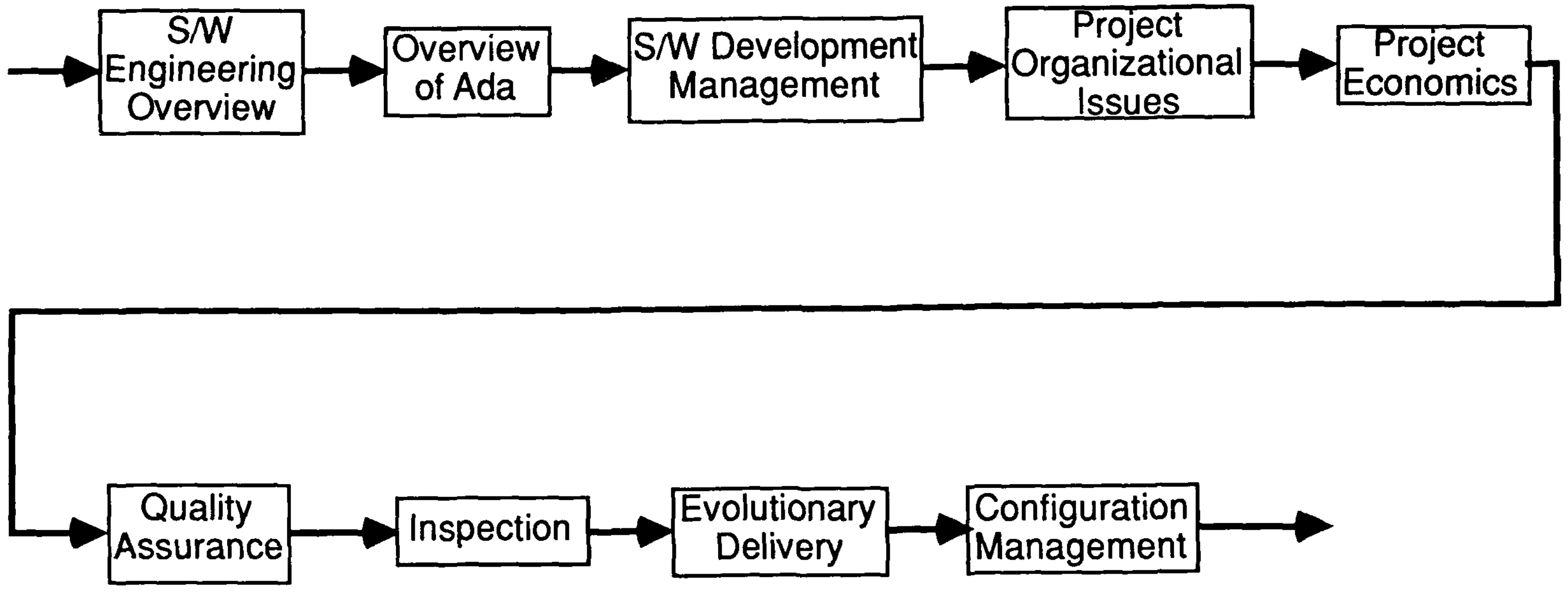
Suggested Curriculum Ada Development


Basic Ada Development Curriculum
Introduction to Ada
Extended Introduction to Ada
Data Structures
Object Oriented Design
Ada Programming Standards
Software Testing & Inspection
Advanced Ada Development Curriculum
Advanced Ada Programming
Advanced Testing
Designing for Reusability
Real-Time Programming/ Tasking Paradigm

Suggested Curriculum Software Engineering



Suggested Curriculum Managerial & Support



Introduction to Ada Outline

Class	Lecture Topics	Lab/Homework
1	**Ada History & Overview** **Simple Subset of Ada** **Introduction to Text I/O** **Introduction to «*Blank*» Compiler & Environment**	Square Root Sin, Cos, & Integration Simple Queue
2	**Abstraction, Types & Subtypes**	Type Questions Advance Queue Driver Abstract List Type Abstract Queue Type Abstract Personnel Information Type Abstract Vector Type
3	**Modularity, Localization & Packages**	Complex Type Package List Type Package Matrix Type Package Personnel Info Type Package Trigonometric Package Queue Type Package Vector Type Package
	Information Hiding & Private Types	Private Complex Type Private List Type Private Personnel Info Type Private Queue Type

Introduction to Ada Outline (cont)

Class	Lecture Topics	Lab/Homework
4	**Scoping & Overloading** **Program Structure**	*Separate Compilation of:* Complex Package List Package Matrix Package Personnel Info Package Queue Package Trigonometric Package Vector Package
5	**Exceptions** **Using Generics** **Text I/O & Exceptions** **Conclusions**	*Exceptions with:* Complex Package List Package Matrix Package Personnel Info Package Queue Package Trigonometric Package Vector Package

If time permits, assign a general problem which requires application of all of the above features of the language and one or more of the packages that the students have developed.

Advanced Ada Programming Outline

Class	Lecture Topics	Lab/Homework
1	**Advance Types**	Parameterized Queue Type Parameterized List Type Variable Field Personnel Type Linked-List List Type Linked-List Queue Type Binary Tree Type
2	**Renaming Declarations Generics & Reusability**	Generic Integration Function Generic Complex Type Package Generic List Type Package(s) Generic Matrix Type Package Generic Queue Type Package(s) Generic Tree Type Package(s) Generic Vector Type Package Generic Interpolation Table Pkg Generic Set Type Package Generic Trigonometric Package
3	**Tasking**	Queued Message Passing Simple Tasking Control - Juggling, Part 1
4	**Tasking (cont)**	Queued Message Passing - Part 2 Simple Tasking Control - Juggling, Part 2

Advanced Ada Programming Outline

Class	Lecture Topics	Lab/Homework
5	**Low Level Features of Ada** **Low Level Features using** *«Blank»* **Compiler**	Simple Tasking Control - Juggling, Part 3
6	*Additional Selected Topics*	**Additional Program**[1,2,3] Cruise Control/Car Simulation Interceptor Simulation Radar/Sonar Simulation

1 These problems are highly system dependent

2 Or some other general problem applicable to the project/company of the student which requires application of all of the above features of the language and one or more of the packages that the students have developed.

3 If time permits.

How Small Companies Can "Develop a Complete Program"

You don't have to be a big company to set up/pay for such a program

- Universities could provide Extension Programs

- Small companies can hire Consultant-Trainers
  - *In cooperation with other companies?*

Participation in Government Sponsored Training Programs

- California Education Training Program (ETP)
- *others??*

California Eduction Training Program (ETP)

Started about 4 years ago

Initially intended to retrain workers (factory) about to be laid off.

Rockwell convinced state they would have to lay off Engineers if were not trained in Ada

• Trained:	450 people
• State Paid:	$2,300/student

State Redesigned Program to accommodate High-Tech training

- Good way of attracting/keep high-tech businesses/jobs
- At least 5 other companies are taking advantage of program

SAIC	Cubic	TRW
Hughes	Northrup	McDonnell Douglas

At Least 2 organization can assist in getting funds:

California Manufacturers Organization
Orion Group

California ETP Requirements

Each student must get ≥ 100 hours of “class”

- Must complete:

Length of Program	%
100-120	90
≥ 120	80

- **State does check.records**

Instructor's & Course Content must be “Approved”

- Prefer California based Training Organizations

Some spending restrictions

e.g. Cannot buy equipment

Conclusions

The **vast majority** of the current programming population trained to be proficient in Ada programming & Software Engineering

- But not necessarily to the same level of ability!
- It will take time

Train your employees **before** they need the knowledge!

Select, Purchase & Train your employees in support tools & software before your project begins

Consider this Ada/Software Engineering Program the start on a continuing process of Maintaining/Improving your competitive edge

Remember: Your employees are the most valuable asset of your company!

- Training is an investment with a high payoff

This Page Left Blank Intentionally

Ada Software Development for Engineers and Executives

Don Thompson, Ph.D.

Computer Science Department
Pepperdine University
Malibu, CA 90265
and
Technical Research Associates, Inc.
760 Las Posas Road, Suite A-4
Camarillo, CA 93010

Abstract

This paper is about a sequence of four modules in Ada which I have developed and currently teach to engineers & managers at the Pacific Missile Test Center at Point Mugu, California, as well as at Unisys, Eaton, and Northrop Aviation in Ventura County, California. The paper details the contents of each of the courses and describes how to successfully implement them, both with respect to preparing and screening prospective students as well as discussion of effective teaching methods. Finally, words of wisdom gleaned from experience are included about teaching software engineering to both experts and novices through the Ada medium.

The courses discussed are 1) *Ada: an Executive/Management Overview*, 2) *Introduction to Ada*, 3) *Intermediate Ada: Software Engineering*, and 4) *Advanced Topics in Ada*. The first course is an 8 hour summary of the main topics and features of Ada, targetted for technical managers in aerospace who seek literacy in the capabilities of Ada as a tool for software development. The last three courses form a sequence aimed at engineering professionals who work in the aerospace industry. The average participant has programmed extensively in Pascal, C, or FORTRAN. The goal of this sequence is to teach "Ada, the Software Engineering Tool" and not simply "Ada, the language". The three course sequence consists of 120 classroom hours, with approximately 75 hours of hands-on programming in Ada.

1. Introduction - The Vacuum in Ada Training

Despite sweeping Department of Defense mandates about the usage of Ada, private industry as well as some armed forces installations (such as the Pacific Missile Test Center at Pt. Mugu, CA) and many University Computer Science Departments have been lethargic about jumping into the Ada scene. Many have taken the 'wait and see' attitude toward adoption and implementation of Ada, partly due to the disappointing availability of compilers and program support environments. (Indeed, the first validated PC compiler did not become available until the fall of 1987.) It is no surprise, therefore, that Ada training has experienced identical lethargy both in development and adoption. The four courses discussed herein were created to help fill the training void.

The need for Ada training avails an opportunity to do more than simply teach/learn another language. Here stands an open door through which sound software engineering education may enter. This is an opportunity which cannot be minimized. The DoD directive has created a captive audience! Instruction should accomplish more than 'code conversion/translation' wherein one learns to simply cast a FORTRAN or C program in an Ada syntactic form. We can make students better programmers, which is ultimately the most important issue. A discipline for the largely undisciplined.

2. Ada: An Executive/Management Overview

Before an organization decides to place a branch office in a foreign territory, its executives must evaluate the terrain based upon a preliminary information survey. Similarly, Ada training must begin or at least include a 'literacy' portion for the managerial level of any software development group. The course *Ada: An Executive/Management Overview* is aimed at providing this literacy; a 'crash course' in the structure, capabilities, and applications of Ada. *By way of analogy to learning a foreign language, this course is primarily concerned with the scenery in the foreign country, essentially at the level of handing out travel brochures.*

Course Objective: The student will be given a solid overview of the history, structure, scope, and applicability of the official programming language of the DoD: Ada. The student will be given a foundation in Ada terminology and concepts which will enable him to decide the applicability of Ada to his needs. Although this course is designed for Executives/Managers, it is also highly recommended for students planning to take the follow on course: Introduction to Ada.

The course carries no prerequisities, although realistically, most of the participants are managers of software groups with extensive programming exposure and background. The typical student enters the course with an "I'd like to check this language out" attitude. At the conclusion of this one day, 8 hour session they have enough primary information about the language to be able to assess its capabilities and appreciate its strengths and weaknesses.

Course Content and Outline: The course proceeds from straight lecture and discussion of the following topics

1. The Development of Ada: Past & Present

2. APSE : The Ada Programming Support Environment

3. Compiler Survey

4. Survey of Major Language/Design Feature
    a. Overall Structure
    b. Logic Constructs
    c. Packages
    d. Subprograms
    e. Tasks
    f. Error Handling
    g. Generics
    h. Separate Compilation

5. Comparison with other High Level Languages

6. Interface with other Languages

7. Sample Applications

8. Embedded Systems - Real Time Programming

9. Programming in the Large

References: This course uses no textbook, but relies exclusively on handout material which includes: selected articles from the Communications of the Association for Computing Machinery, an Ada glossary, a list of Ada Language Textbooks, summaries of selected articles from the Ada Information Clearing House, selections from the DoD-STD-2167A Military Standard on Defense System Software Development, and numerous programming examples developed by the author.

Typical Application: One of the many items handed out in this course is a list of current software projects being developed in Ada in the commmercial sector, the military environment, and the world of academia. The students are reminded that much has been done and is being done in Ada. For example, the Planning Research Corporation (PRC) of Houston has assembled a top flight team of computer professionals and is currently seeking Ada programmers to assist in the development of the Space Station Software Support Environment, a fully capable Ada system. PRC is responsible for the implementation of the process control, database management, communications, and real-time reporting functions for the first U.S. "software factory" which is being used to develop flight and ground software for the Space Station Program. They are seeking Ada programmers with at least 3 years of experience.

Words of Wisdom: The course can be deemed a success if managers leave with a positive attitude toward and appreciation of the language or better yet a desire to take the Introductory Ada course and beyond.

3. Introduction to Ada

The introductory course is truly the cornerstone of this four course curriculum. It therefore requires careful student screening. Moreover, thoroughness, rigor, and attention to detail in coverage and presentation are required. *We are no longer merely leafing through the travel brochures. Indeed, we have purchased our tickets, obtained all of our vaccinations and travel sickness medication, and have enrolled in the intensive language school for our visit to the foreign land.*

For this course, the emphasis is not on software engineering - at least, not yet. Students are still learning syntax and the Ada programming style. *By analogy, one does not become a part of another foreign culture after one brief visit. It takes a certain amount of indoctrination; time spent learning the ways and customs.* Most programmers begin with this course, skipping the executive overview.

Course Objective: This course will be a hands-on introduction to Ada. The student will learn how to use the major featrues of this DoD developed language with a PC-AT based Ada compiler (DoD validated). Upon completion of this course the student should be able to program in Ada. Additionally, the student will be able to make an informed decision about how well a particular application is suited for implementation in Ada.

Course Prerequisities: Proficiency in (at least) one of the following languages: Algol, C, COBOL, FORTRAN , Pascal, or PL/1. The instructor will meet with prospective students no later than two weeks prior to the beginning of class in order to assess the class level and answer any questions about course content.

Student Level Assessment Questionnaire: Despite published course syllabi and course prerequisites, many students enroll in this course with inadequate preparation. Thus, a pre-course screening process has been developed. At least two weeks prior to the offering of the course, the instructor meets with prospective students to discuss the course layout. The following questionnaire is filled out by the prospective students and serves as a benign means of screening out weak or poorly prepared students as well as giving the teacher feedback on the background and expectations of qualified students.

1. What is Structured Programming?

2. What is Top-Down Design?

3. How much programming have you done in Pascal or PL/1 or C or Modula-2? (Estimate the number of terminal hours.)

4. Have you ever written a recursive subprogram? If so, what did it do?

5. What is a record?

6. What is a pointer?

7. What is a linked list?

8. With what high-level language are you most familiar? (Estimate the number of terminal hours.)

9. What is Bottom-Up Design?

10. Briefly describe your interest in learning Ada. What are your expectations?

Response Statistics:

Responses to question #3 range from "thousands of hours of C" to "I took a course in Pascal about three years ago". As for question #8, the two most commonly listed languages are FORTRAN ("10 billion hours" according to one respondent) and BASIC. Most of the responses to question #10 cite the need to learn Ada for programming on DoD contract work and the ability to read or understand Ada code without necessarily having to write programs. Several students enter the course with the attitude that they are coming "just to pick up another language", as in rock collecting. Few see past the "translation/conversion" level of Ada. That is, most want to simply be able to convert their FORTRAN or C programs into Ada syntax. Few are interested in learning about the software engineering aspects of Ada.

Course Content: The course will begin with an overview of the major capabilities of Ada; then the language will be presented as a series of stepwise refinements covering, but not limited to, the following topics: Ada and software engineering, overall Ada style, types (enumeration, numeric, array, characters and strings, discriminated, record, access, private), control statements, subprograms and parameter passing, packages, exceptions, generics, separate compilation, i/o, tasks and concurrent programming. In addition, programming examples will be carefully examined. The student is expected to complete a number of programming lab assignments which will re-emphasize those topics discussed in lecture.

Course Outline: The course runs for 40 hours. Approximately 60% of this time is devoted to lab work which consists of hands on programming of Ada exercises on PC-AT machines.

Day 1	(am)	Historical Background, Recent Developments, Ada Overview, the APSE Lab
	(pm)	Ada Program Structure, Notation, Discrete Data Types, Statements Lab
Day 2	(am)	More Statements, Declarations & Blocks Lab
	(pm)	Subprograms - procedures & functions, Packages Lab

Day 3	(am)	More Packages, Structured Data Types Lab
	(pm)	Program Structure, Access Types Lab
Day 4	(am)	Tasks & Concurrent Programming Lab
	(pm)	Exceptions Lab
Day 5	(am)	Generic Units Lab
	(pm)	File I/O: Direct, Sequential, Text Lab

References: The textbook for this course is An Introduction to ADA (2nd Ed.), by S.J. Young; Wiley Publishing. Young's work is very readable and has a rich variety of different levels of Ada programming examples ranging from program fragments to subprograms to full blown application programs at the end of each chapter. In addition, he provides an appendix containing solutions to many of the programming exercises. Finally, many handouts are provided by the instructor, giving additional examples and historical background.

Typical Application: The following lab is given to the students on the afternoon of the third day.

SPARSE POLYNOMIALS

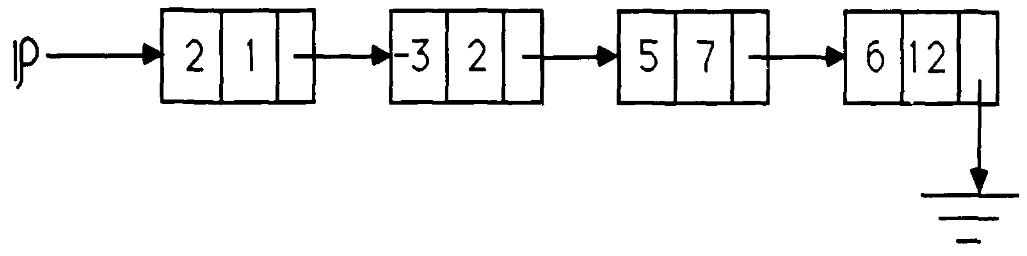
One common application of pointers is to create linked lists to represent sparse objects. For example, the polynomial

$$P(x) = 2x - 3x^2 + 5x^7 + 6x^{12}$$

could be represented by a coefficient array:

$$P = (0,2,-3,0,0,0,0,5,0,0,0,0,6)$$

or more efficiently by a linked list:



Write a procedure POLYNOMIAL which takes a coefficient array and creates a linked list for some polynomial P(x), returning a pointer to the first node of the list. Embed your procedure in a main driver which asks the user for an integer value "a" and then prints P(a).

Bells & Whistles: Write a procedure ADD which takes two linked list polynomials and forms their sum. Experiment with the use of a circular linked list for each polynomial.

Words of Wisdom: As the instructor of this course, one feels a bit like the salesman who is trying to sell a refrigerator to an eskimo. The product has not received wide acceptance. It is as if the teacher is there to **defend** Ada *before the* (sometimes hostile, often pessimistic) FORTRAN/C/Pascal fan clubs. The choice of a programming language is an emotional issue for some. Ada language training must therefore be given at a level beyond syntax and semantics. The instructor must convey Ada's software engineering and intellectual strengths in order to overcome the learning barrier erected by some programmers. Only then will they embrace it.

4. Intermediate Ada : Software Engineering

*We are no longer tourists. We have taken up temporary residency in this foreign land. We are beginning to **think in Ada.***

Course Objective: This course will be a hands-on continuation of the Introductory Ada course. All students will use a validated DoD compiler for the PC-AT. Upon completion of this course the student will be able to program with 100% of the features of ADA using sophisticated data structures and software engineering principles in the programming application areas of science (engineering) and data processing.

Course Prerequisites: 6 months programming experience in Ada or completion of the Introductory Ada course.

Course Content: The course will emphasize advanced data structures and software engineering concepts in Ada through the following topics: access types, linked lists, records, discriminated data types, private types, limited private types, generic program units, function parameters, packages, separate compilation, recursion, direct i/o, sequential i/o, error handling techniques, tasks, conditional and timed entries, delays, timeouts, task priorities, concurrent programming, and low level programming. The student will be expected to complete several medium length programming lab assignments which will re-emphasize the topics discussed in the lectures.

Course Outline: The course runs for 40 hours. Approximately 60% of this time is devoted to lab work which consists of hands on programming of Ada exercises on PC-AT machines.

Day 1	(am)	Review of Fundamental Ada Features Lab
	(pm)	Data Encapsulation & Abstract Data Types Lab
Day 2	(am)	Discriminants & Linked List Applications Lab
	(pm)	Recursive Data Structures & Recursion Lab
Day 3	(am)	Concurrency: Fundamental Issues Lab
	(pm)	Concurrency: Conditional & Timed Entries, Delays, Time Outs, & Task Priorities Lab
Day 4	(am)	Exceptions & I/O Lab
	(pm)	Generic Facilities & Subprogram Parameters Lab

Day 5 (am) Representation Clauses, Low Level Programming, & Interrupts
Lab

(pm) Program Structure, Separate Compilation, Pragmas, Interfacing with other languages
Lab

References:The textbook used for this course is ADA - An Advanced Introduction, by N. Gehani, Prentice-Hall Publishing. Gehani's book is a joy to read. He has compiled a rich array of non-trivial Ada programming applications, illustrating the full depth of the language and providing for many excellent examples of sound software engineering practice. Handouts are also provided.

Typical Application: The following lab assignment is given to the students on the morning of the third day.

OBJECT ORIENTED DESIGN: AN ELECTRONIC MAIL SYSTEM VIA TASKS

A fixed set of N Producer processes puts messages in a mailbox of N single element slots. Process P(i) always places its messages in the ith slot. A single Consumer process removes the messages in a round robin fashion (meaning that if the mailbox is viewed as a circular buffer, the Consumer process goes around taking messages out of non-empty slots moving in a fixed direction, either clockwise or counter-clockwise.) A Producer should not overwrite its own messages and the Consumer should not try to remove a message before it has been delivered.

Write an Ada program to implement this electronic mail system.

Bells & Whistles: Have each mailbox be capable of holding a fixed number (> 1) of messages in a circular buffer. Allow each producer to send messages to more than one mailbox. Have the consumer read an arbitrary number of messages per box with the number being determined via a random number generator or via a master control task.

Words of Wisdom: This course is a pleasure to teach. By the time a student reaches the conclusion they will have a full command of the language and will be able to produce quality software. All that remains is the need for practice with the Ada medium.

5. Advanced Topics in Ada

This is primarily a hands-on, instructor-off course. (The best teacher is often one's self or another student.) *The tourist turned temporary visitor has become a resident of the foreign (now domestic) nation.* Students are given ample lab time to polish their Ada programming skills, to work in small groups with other students, and to modify code produced by other teams of programmers. One reason for the existence of this course is that many contracts require a minimum of 100 hours of Ada training on the part of the programmers involved, so many engineers have found the need to take all three courses in the programming sequence. In all honesty, by the time the intermediate course has been completed, the student is proficient. The advanced course merely provides the opportunity for structured practice.

Course Objective: This course will be a hands-on continuation of the Intermediate Ada course. Upon completion of this course the student will be an expert Ada programmer, able to write sophisticated Ada applications software either individually or in a team environment.

Course Prerequisites: 1 year programming experience in Ada or completion of the Intermediate Ada course.

Course Content: This course emphasizes Ada 'programming in the large', object-oriented development, and software reusability. The examination of several case studies of large Ada programming systems is provided. In addition, the student's lab work will involve 1) modifying and expanding several large Ada application packages, thus gaining an appreciation for the issues of software maintenance and software reusage in Ada; 2) participation in the Ada software engineering cycle by designing, coding, testing, and implementing a large Ada Application in a small group.

Course Outline: The course runs for 40 hours. Approximately 60% of this time is devoted to lab work which consists of hands on programming of Ada exercises on PC-AT machines.

Day 1	(am)	Ada & Object Oriented Programming
		Lab
	(pm)	Advanced Data Structures
		Lab
Day 2	(am)	Knowledge & Representation Issues
		Lab
	(pm)	Graphs
		Lab
Day 3	(am)	Programming in the Large
		Lab
	(pm)	Reusage of Modules
		Lab
Day 4	(am)	Tasking: Multiple Threads of Control
		Lab
	(pm)	Deadlock
		Lab
Day 5	(am)	Fundamental Task Design & Advanced Tasking Concepts
		Lab
	(pm)	Scheduling & Optimization
		Lab

References: The following two textbooks are used: Software Components with Ada, by G. Booch, Benjamin-Cummins Publishing; Real-Time Ada Workbook, by Softech, Center for Tactical Computer Systems. Booch's book is an excellent comprehensive work about Ada 'programming in the large', software reusage, and object oriented design. The Softech book contains interesting real-time applications and provides for a thorough coverage of non-trivial tasking concepts. Handouts are also provided.

Typical Application: The following lab is given to the students on the afternoon of the second day.

NETWORK UTILITIES

The package Network_Utilities listed in Ch. 12 of Software Components with Ada, by G. Booch contains a routine that finds the shortest path between any two vertices of a network. Modify the package so that it is as generic/reusable as possible.

Bells & Whistles: Use your generic package to generate a minimum distance rooted spanning tree for a given network.

Words of Wisdom: If students get to this point, they can begin teaching their own Ada training sessions!

6. Conclusions

Effective Teaching Methods: The use of an overhead projector with transparencies containing all textbook examples and handouts is highly recommended. Anything that goes up on the screen should already be in the student's hand. One need not force the students to waste time with the art of *secretarial note-taking.*

Without a doubt, these last three courses need to be hands-on. Indeed, were it not for the shortage of time and the somewhat threatening nature of having to sit at a terminal amid other students, the executive course could benefit from some lab time as well.

The screening process for the introductory course is highly recommended. It makes the course run much more smoothly, enabling the instructor to demand (and get) more from the students.

This instructor usually includes copious hints and occasionally the "shell" of a program (to help get things off the ground) for the labs. Exercises with multiple tiers of difficulty for multiple levels of ability are also very useful. It helps to be flexible with the syllabus, enough so to allow for all of those "what if" types of questions. Indeed, such questions are invited.

Finally, enthusiasm about the subject matter breaks down many barriers and keeps student interest and optimism at a peak.

This Page Left Blank Intentionally

Integration of Ada Software Engineering Training within the Ada System Development Process

June 16, 1988

Third Annual

Ada Software Engineering Education

and Training (ASEET) Team Symposium

Presented by:

Mr. Kim Hoover
Planning Analysis Corporation
8400 West 110th Street
Overland Park, Kansas 66210
(913) 451-8833

INTEGRATION OF Ada SOFTWARE ENGINEERING TRAINING WITHIN THE Ada SYSTEM DEVELOPMENT PROCESS

Mr. Kim L. Hoover
Planning Analysis Corporation

In July of 1987 Planning Analysis Corporation (PAC), acting under contract to the United States Marine Corps, was tasked with analyzing a series of issues associated with Ada. Among these issues were the impact of Ada on current Marine Corps standards, methodologies available to support Ada development, and tools available to support design and programming of Ada software. The report subsequently delivered represents a rigorous review and evaluation of what PAC found to be the key issues facing organizations undertaking the transition to Ada technology. Our findings demonstrated several clear facts. First, the true impact of Ada is best understood when viewed in its totality as a software engineering process rather than a programming language. Second, the process of system development in Ada requires clear and systematic interface between the analysis, design, and programming phases of system development in order to work effectively. And third, adequate training programs are required in order for the Ada process to become viable as a system development alternative.

The contents of this paper are presented in two parts. The first consists of extracts taken from the analysis conducted by PAC for the Marine Corps on Ada. This will lay the groundwork for understanding our view of the Ada process and serve as the foundation for part two. Part II outlines a training approach developed by PAC to support the Ada software engineering process. This approach is embedded within Planning Analysis Corporation Ada Group Enterprise (PACAGE), an organization specifically designed to support both Ada system development and Ada technology transfer. PACAGE was created in response to our recognition of the need for combining Ada education within the project development environment. We believe PACAGE to be an innovative, efficient, and cost-effective means for organizations to gain the knowledge and skills necessary to adequately pursue system development in Ada.

PART I

SECTION 1

REQUIREMENTS FOR SOFTWARE ENGINEERING LEADING TOWARD IMPLEMENTATION IN Ada

The core activities of software engineering within the life cycle are analysis, design, and programming. Discussion in this section addresses the requirements for these activities when associated with program development in Ada.

ANALYSIS

This section concentrates on issues associated with analysis conducted in support of an Ada development effort. The term "analysis" here refers to the examination and subsequent documentation of system requirements. The purpose of the analysis phase is to produce a clear and concise functional specification. This is commonly referred to as a description of "what" functions the system must support, independent of "how" they are implemented. Within the popular development methodologies, i.e., Structured Analysis and Structured Design, this analysis is documented by production of a functional specification in the form of data flow diagrams, a data dictionary and process specifications. This analysis method, as tailored and popularized by Ed Yourdon, generates the Functional Requirements Definition (FRD) to complete the functional analysis of a system.

Investigation of Ada and the analysis phase of the life cycle indicates no requirement for alteration to traditional methods. The need to develop a functional user system specification, independent of implementation considerations, remains constant for Ada as for other programming languages. The use of Ada neither provides nor demands special consideration for the analysis effort. Grady Booch, author of Software Engineering with Ada, supports this view:

> It would be inappropriate to even think of using the Ada language at this point in the life cycle. The reason is simply that the language is part of the solution, not the problem space. Analysis should be accomplished independently of any implementation language, and our use of Ada at this point would be premature. (Reference b, p. 415)

In conducting the Marine Corps analysis PAC studied various system development methods. Among these were Jackson System Development, Yourdon Structured Analysis and Design, Object Oriented Design, and Pictorial Ada Method for Every Large Application (PAMELA 2). Each of these approaches was found to contain a phase equivalent in nature, although differing in structure, to the Yourdon FRD. The need for documented functional system specifications is a key theme in most modern system development methodologies.

Most of the available documentation relating to Ada discusses the use of the programming language itself. The second category of information available deals primarily with design techniques that support Ada development. Among these writings are references to analysis approaches that could be used. Booch makes the following observation:

> Keep in mind that object-oriented development is a partial life-cycle method; it focuses on the design and implementation phases of software development. As Abbott observes, "although the steps we follow in formalizing the strategy may appear mechanical...[it] requires a great deal of real world knowledge and intuitive understanding of the problem"[2]. It is therefore necessary to couple object-oriented development with appropriate requirements and analysis methods in order to help create our model of reality. We have found Jackson Structured Development (JSD) to be a promising match. General data-flow techniques, such as those by Gane and Sarson, are also useful tools in building this abstraction of reality. (Reference b, p. 47)

The available data indicates that the choice to implement a system in Ada has little to no impact on the analysis phase of the life cycle. The fact that Ada is the target language stands apart from the choice of a particular method to support the analysis function. Other considerations come into play when deciding which analysis technique to adopt. This decision centers on issues relating to training, availability in the public domain, organizational needs, and consistency with follow-on activities.

DESIGN

This section presents an overview of the implications of Ada in software design. The term "design" in software engineering refers to the activity that produces a specification for the software being developed. This activity is normally documented via a systematic application of textual and graphic representations. Unlike analysis, the design phase of development documents the "how" aspect of the automated solution to the problem. Beginning on the macro level and working steadily toward detailed specification, design becomes increasingly environment and language dependent. The discussion in this section presents an overview of the requirements for design generated out of a decision to use Ada as the implementation language.

Following analysis and preceding actual design, Booch discusses what he calls the Requirements Definition Phase. He states:

> The purpose of this phase is to take the allocated functions from the system specification and determine the detailed requirements for the software. ...The primary product of this phase is an approved development (functional) software specification. (Reference b, p. 416)

Within the framework of the life cycle this phase equates to development of the a new physical model. It is within this phase that the elements of analysis and design blend into a transition activity. The transition that takes place here initiates the design continuum that culminates in the completed software design. Booch points out that "it is very difficult to refrain from trying to design a complete solution at this point, but we must stay at a higher level of design." (Reference b, p. 416) To some degree the end of requirements definition and the start of actual design requires intuitive recognition. The line of demarcation often associated with this milestone is the production of an initial system design of commonly agreed upon detail.

Based on the information gathered, the decision to implement in Ada begins to impact with the development of this initial system design. Booch comments:

> At this point, we shall assume that we have chosen Ada as the design language. Since one product of this phase is an initial design, we may use Ada program unit specifications to document the highest level of our system. (Reference b, p. 416)

George Cherry, developer of the PAMELA 2 design method, also identifies the step after analysis (which he calls requirements specification) as follows:

> <u>Develop the Hierarchical PAMELA Graphs (HPGs)</u>
> In this phase, software engineers use the Pictorial Ada Language (PAL) and guidelines of PAMELA 2 to transform the software requirements specification (however expressed) into a PAMELA 2 design description. (Reference c, p. 102)

Both cases reflect the position that design, even at the highest level, is language dependent. Accordingly, it appears best to adopt a design method that supports the implementation characteristics of the language at the very start of the design activity. This answers the question of when the transition to an Ada design technique should take place.

The inquiry now shifts from identifying the point at which Ada impacts on the design effort, to one of determining what the requirements are for designing toward Ada implementation. The answers to this lie in understanding the philosophy behind the development program that produced Ada. A good summary of pertinent background information is found in the preface of a text produced by EVB Software Engineering, Inc. entitled <u>"Analysis and Design for Ada."</u> The following excerpt from that text is provided as a basis for understanding the impact Ada has on the design function.

> Not only is the DoD starting to mandate the use of Ada, it is also concerned with the manner in which Ada is used. Ada itself was designed to meet the well-recognized software engineering goals of:
>
> o modifiability,
> o efficiency,
> o reliability, and
> o understandability.

Accordingly, it is the first programming language which both encourages, and enforces, the software engineering principles of:

- o abstraction,
- o information hiding,
- o modularity,
- o localization,
- o uniformity,
- o completeness, and
- o confirmability.

Thus the DoD is looking at software development methodologies which will:

- o ensure that the design principles of the language are fully exploited in order to achieve these software engineering goals,

- o ensure that this large, powerful language is not abused, and

- o help developers handle the ever increasing complexity of software systems.

(Reference a, p. PREF-1)

As earlier noted, the design phase of software development is dependent on the language and the implementation environment. From the preceding summary, it is clear that the intended use of Ada within DoD places an additional level of requirements on the design phase of the software development. In fact, the phenomenon of Ada is more closely interwoven with the design emphases cited than with the mere use of the language as a coding alternative. Independent of the software engineering concepts listed above, it would be unlikely that the benefits associated with Ada could be maximized.

From its inception Ada was specifically designed by DoD to support, and in some cases require, the incorporation of advanced software engineering concepts. Booch calls attention to the fact that "In a sense, Ada is a language that directly embodies many modern software engineering principles and is therefore an excellent vehicle with which to express programming solutions." (Reference b, p. 4) The conclusion derived from this background material is straightforward. The decision to adopt Ada as the implementation language for application software carries along with it the requirement to incorporate a complete engineering philosophy into the development effort. The requirement then exists to find and use a design approach that supports this philosophy.

PROGRAMMING

From the standpoint of actual programming, Ada levies no new requirements on the software development process. To a large degree the rigorous application of software design techniques associated with Ada development reduce the level of activity normally associated with programming. As Booch notes:

> With Ada chosen as the implementation language, this phase is relatively simple and is actually an extension of the process started during design. At this point, we should have a detailed design, represented by a set of Ada program units with complete specifications. During the coding phase, we need only complete the implementation of our unit bodies." (Reference b, p. 419)

This situation is entirely consistent with the philosophy under which Ada was developed. By taking into consideration the implementation aspects of the language during design, most programming decisions are made prior to actual coding. This approach supports in practice the software engineering goals and principles referenced above. Beyond this, standard practices associated with good programming techniques are in order.

SECTION 2

REQUIREMENTS FOR TRAINING IN SUPPORT OF Ada

The goal of Ada development within DoD has been to improve productivity while achieving system reliability and adaptability. The intent is to make development and support faster, less expensive, and more predictable, resulting in software that is more powerful, reliable, and adaptable. One of the stated objectives leading toward this goal is the improvement of personnel resources through increasing the level of expertise and expanding the base of expertise available to DoD. The mechanism by which this objective will be met is increased emphasis on technology transfer. Among the primary activities that will contribute to this technology transfer is adequate training of personnel.

There are three categories of personnel required to successfully develop a software project through the development phases. These are the functional analyst, the designer, and the programmer. A specific set of skills is associated with each of these roles and training must be provided to develop those skills. The following summary briefly describes each role and outlines the training that will be required in order to transition to an Ada development environment.

Functional Analyst

The role of the functional analyst is one of problem definition. The primary product produced by the analyst is the functional system specification, or the "WHAT" view of the system. As previously observed this function is not language dependent and does not require implementation oriented expertise. Rather, training for analysts should focus on the particular documentation method chosen to represent the analysis conducted. A lesser degree of training in the selected documentation method is also required for personnel reviewing the functional specification.

Designer

The role of the designer is to take a completed functional system specification and transform it into a system design. To this end designers require the highest levels of expertise of anyone in the development process. Accordingly, they also require the highest levels of training. First, the designer requires training in the analysis method adopted within the framework of the methodology. This ensures a communication vehicle between the functional analyst and the system design team. Design is an iterative process that requires review of the functional specification as additional information is discovered during design. Clear communication between the analyst and the designer is critical to successful system development.

Next, the designer must be trained in the language and operating environment in which the system will be implemented. As was noted earlier, design is very much language dependent. Therefore, a transition to Ada demands that designers receive thorough training in Ada. This is particularly true because of the many unique constructs and characteristics of Ada. In most cases it will take formal training followed by a period of practical experience to become an experienced Ada programmer. Beyond this, additional levels of application experience are required to assume the role of system designer.

The third area where the designer requires training is in the design method selected for the development effort. This includes training in the techniques as well as the tools adopted in support of the design approach. The key element involved in design lies in the dynamic process of combining the requirements of the functional specification, with the constraints and capabilities of the implementation environment, to generate a viable system. Because of the obvious high demands placed on the design, a team approach is often adopted to ensure all of the necessary skills are present when required.

Programmer

The skills required of the programmer are twofold. First, they must be trained in the programming language itself. This training is available from various sources and in various forms. Classroom instruction, computer-aided instruction, seminars, and on-the-job training are all contributors to learning the language. It is important to recognize that learning to program is an evolutionary process. In their proper place, any or all of these training alternatives should be used to facilitate programmer enhancement. Adequate training, along with practical experience, can help to maximize the programmer resources available.

The second area where the programmer requires training is in the design techniques being used. The programmer needs to understand and interpret the design documentation. Although this may require a lesser degree of training than for designers, it is critical to success. The communication between the programmer and designer is as important as that between the designer and the analyst. The philosophy of Ada relies heavily on the ability to accurately represent the problem in the programmed solution. The programmer's ability to accomplish that task is dependent upon good communication with the system designer. The vehicle for the bulk of that communication is the design document itself.

PART I SUMMARY

It is clear from a review of PART I that making the transition to an Ada oriented system development process represents a major undertaking within DoD. In a sense Ada requires a revolution in our thinking and practice associated with software development. Adoption of sound engineering principles is a prerequisite, and mastery of the life cycle steps as an interrelated whole lies at the foundation of the Ada process. These requirements, viewed in conjunction with the spectrum of organizations within DoD faced with implementing systems in Ada, call for creation of training mechanisms adequate to meet the demand.

Another of the points made clear from PART I is that the selection of a design methodology lies at the center of any decision to pursue Ada. As a programming language, Ada realizes its true potential when used to implement software designs that reflect software engineering principles as established by DoD. To this end PAC has selected PAMELA 2 as its design methodology. However, the presence of a methodology is only a part of the overall picture required to successfully produce systems in Ada. In response to the recognized need for a comprehensive approach to incorporating Ada within DoD organizations, PACAGE was established. PART II of this paper outlines the philosophy and capabilities of PACAGE to serve as an effective vehicle to accommodate Ada system development and Ada technology transfer for DoD organizations. The integration of Ada training and development expertise into a single operating entity forms the basis of the PACAGE concept.

PART II

PLANNING ANALYSIS CORPORATION Ada GROUP ENTERPRISE (PACAGE)

Capabilities Summary

BACKGROUND

The development of Ada by DoD began in earnest in 1975. Since that time a tremendous level of effort has been expended in the development of the language, compilers, design methodologies, and assorted support tools. From the start of work on Ada the expressed intent and expectation was to produce a language that would help reduce the cost associated with software development. But in spite of the thousands of man-years spent in developing Ada to its current state, this goal remains substantially unfulfilled. This was vividly underscored by Army Undersecretary James Ambrose in his address to the 1987 AdaExpo held in Boston. Ambrose directly made his point by saying, "I am not impressed with the qualitative measures of superiority, productivity or virtues of Ada." (Reference d, p.33) A long time and continued supporter of Ada, Undersecretary Ambrose raises a key issue associated with the future of Ada. The question facing the Ada community today is, where are the results and benefits this language is designed to produce? Producing these results is the current challenge for the Ada community.

EVOLVING TECHNOLOGY

The development effort to produce Ada has clearly been the most strategically planned software engineering advancement in history. As a result, proponents of the language consistently point to Ada as a language ideal for modern software development. Many promote the position that this language has the potential to support software that is modifiable, efficient, reliable, and understandable. Furthermore, Ada encourages engineering principles that include abstraction, information hiding, and modularity. Based on a review of its programming characteristics, Ada represents a technology success. However, the language alone will not produce the results sought within DoD. Other components of the software development process must be in place to achieve the productivity advantages envisioned for Ada software. It is to this end that PACAGE has been formed.

ORGANIZATION

PACAGE is a forerunner of the type of organization necessary to support the continued evolution of Ada toward its intended objectives. With the express purpose of providing a full service life cycle based development vehicle for Ada, PACAGE represents the next step toward widespread acceptance of Ada as the implementation language of choice. Combining the skills, tools, methodologies, and training required to support Ada development into one cohesive business entity, PACAGE offers an ideal mechanism for software engineering efforts. In addition, PACAGE has been specifically designed to facilitate transfer of Ada technology into the organization utilizing its services. As Ada moves into the future, PACAGE will continue to monitor new developments and provide the most efficient software production environment available. The goal of PACAGE is to enhance the value of Ada via ambitious and aggressive application of current technology to today's increasing software needs.

PACAGE is organized to bring together in one team all of the various skills and resources required to produce high quality Ada software. Along with system development, PACAGE facilitates the transfer of Ada technology to user organizations via programmed training and practical project experience. These capabilities have been established through the alliance of four independent companies; AETECH, Navajo Technology Corporation, Planning Analysis Corporation, and Thought**Tools, Incorporated. Each member company possesses unique expertise in the various stages of life cycle system development associated with Ada. Three of the team members also have training specialties, two of which specifically address education in the areas of Ada programming and design. The following is a summary description of each PACAGE member company.

AETECH
Solana Beach, California

AETECH, headed by retired Air Force Colonel James T. Thomes, is a company that specializes in both programming, and in training of Ada programmers. Formerly Commander of Air Force Computer Training, Thomes established the first large-scale Ada Training Program at Keesler Air Base, Mississippi, in 1983. Since that time Colonel Thomes has developed a computer-aided instruction Ada Training Environment, a software development Ada Workstation Environment, and a fully automated set of Ada Instructor Courseware. All of the materials developed by AETECH are written fully in Ada. AETECH also provides on-site classroom instruction for Ada. In addition to conducting training, AETECH provides programming services on several Ada projects under contract to DoD.

NAVAJO TECHNOLOGY CORPORATION (NTC)
Navajo Nation, Leupp, Arizona

NTC was incorporated in 1983 within the Navajo Nation as a highly specialized software development firm, organized to meet DoD programming needs for Ada. Since that time NTC has been involved in a number of DoD sponsored Ada projects, with a strong emphasis on software tool development. NTC also has extensive Ada programming experience for various hardware and operating environments, gained from work involving porting of Ada software. Among other accomplishments, NTC has recently released their Professional Ada & Engineering Workstation. This NTC product consists of an advanced 80386 microprocessor workstation combined with carefully selected hardware and software components. As a support tool the Professional Ada & Engineering Workstation represents the leading edge of microcomputer-based Ada development technology.

PLANNING ANALYSIS CORPORATION (PAC)
Arlington, Virginia, and Overland Park, Kansas

PAC was established in 1981 by its current president, Edward C. McConnaughey, Jr. PAC's expertise lies in providing information planning, system analysis, and computer systems development support to client organizations. This is accomplished through applying a balanced combination of technical, analytical, and managerial skills in the system planning and development process. Experts in life cycle management, PAC is staffed with analysts and designers experienced in working with functional users and data processing professionals. Utilizing structured analysis and design techniques, PAC has assisted a number of DoD, governmental, and commercial clients in producing and implementing automated data processing systems. PAC's approach focuses on bringing about the coordination and commitment necessary to ensure success of system development projects.

THOUGHT**TOOLS, INCORPORATED
Reston, Virginia

Thought**Tools was founded by George W. Cherry, developer of PAMELA (Process Abstraction Method for Embedded Large Applications) and the recently released

PAMELA 2 (Pictorial Ada Method for Every Large Application) design methodology. A known and respected authority in the Ada community, Mr. Cherry is considered a pioneer in the formulation of a design methodology that addresses the use of Ada within the context of the software development life cycle. His work is having a major impact on Ada use by providing a methodology that supports clear and consistent design documentation for Ada software. With a wealth of experience in software design and training, Thought**Tools provides design support as well as professional training in the use of the PAMELA 2 methodology.

PACAGE AS A SYSTEM DEVELOPMENT MECHANISM

One of the capabilities of PACAGE is to produce systems in Ada. This can be accomplished by applying the skilled resources of the PACAGE team to a specific software engineering project. Primary candidates for this use of PACAGE fall within the tactical and administrative information handling arena. These would include tactical command and control information systems, administrative support systems (pay, personnel, supply, etc.), and various management information systems. Within the framework of a traditional life cycle approach, the following outline depicts how the skilled resources of each PACAGE company would work together to produce a system.

STRATEGIC PLANNING/INITIAL REQUIREMENTS/ECONOMIC ANALYSIS

PAC would work directly with the client to document the goals and objectives of the proposed system. Following this step PAC would assist in conducting the feasibility study/economic analysis leading toward selection of a conceptual system solution alternative. In performing this work PAC would work with the project managers and designated functional user representatives to develop an initial picture of functional system requirements. PAC would also rely on other PACAGE members, as well as outside experts, to gather necessary information required by decision makers to direct the project. Upon conclusion of this activity PAC would document management decisions and initiate the next phase based on those decisions.

FUNCTIONAL SYSTEM SPECIFICATION

In this phase PAC's seasoned systems analysts would work with users and managers to develop the functional specification for the system. Through application of structured analysis techniques, the detailed functional requirements for the system would be documented. Appropriate automated support tools would be selected and incorporated in support of this effort. Where appropriate the NTC workstation can be applied as the platform for the various automated tools being used throughout this and subsequent stages of development. The emphasis throughout this phase would be open communication with users and management to ensure production of an accurate functional specification. The documentation produced from this activity becomes the primary source of information leading into the system design phase.

SYSTEM DESIGN

Often viewed as the most demanding work associated with computer system development, the design phase requires a combination of skilled personnel. The PACAGE approach to this effort involves development of a highly skilled and experienced design team. This team includes analysts who produced the functional specification, designers experienced in the use of the PAMELA 2 methodology, and programmers who will eventually implement the system design. The exact number and composition of the design team is determined by the scope of the project and the needs of the client. Again, as in analysis, documentation of the design is supported through use of the most current automated tools. The resources to carry out this activity are already resident within the PACAGE organization. Direct access to the capabilities provided by Thought**Tools, Inc. play a significant role in this phase of the life cycle.

PROGRAMMING

To carry out the task of implementing the design in Ada code, PACAGE relies on the expertise of both NTC and AETECH. Here the team concept continues as designers work with experienced professional programmers to ensure clear communication and common understanding of the design. The product of this teamwork is a fully functional system that follows the design, and thereby meets the needs of the user as defined in the analysis phase. Support tools, such as the AETECH Ada Workstation Environment, are used as appropriate to maximize programmer productivity.

TESTING

Within the PACAGE framework, quality assurance and testing begin at the start of a development project. The analysis documentation is subjected to a series of reviews to ensure both technical accuracy as well as functional completeness. Designs are also subjected to close review. With some of the facilities provided by Ada, designs can be subjected to a series of automated tests to evaluate quality. The use of various automated tools assist in this function. Finally, program code is subjected to rigorous module, unit, and integration testing before release.

IMPLEMENTATION

During system implementation PACAGE resources are available to meet the spectrum of client needs. PAC is experienced in development of user oriented materials that include user manuals and user training. With expertise in various hardware and software environments NTC can provide professional assistance in system installation. PACAGE is also equipped to analyze issues associated with making the transition from existing to new systems, and developing carefully thought-out transition plans.

MAINTENANCE

In terms of system maintenance or upgrade, PACAGE is able to bring the same resources to bear as during the development effort. This support can be provided on either a contract or consulting basis, depending on the specific needs of the client.

PACAGE AS A TECHNOLOGY TRANSFER MECHANISM

In support of long-term and large-scale system development, many organizations have a need to internalize the skills associated with Ada software engineering. To facilitate this objective PACAGE provides a full set of training services. A brief description of this training is provided below. Training can be quickly tailored to meet the specific needs of the client.

STRATEGIC PLANNING/INITIAL REQUIREMENTS/ECONOMIC ANALYSIS

In the area of managing computer system development, PACAGE is staffed to provide training in strategic planning, life cycle management, and alternatives analysis. In conjunction with these areas, classes on project management may also be provided. Specifically relating to Ada, a management seminar may be conducted that outlines the history, goals, and current directions of this language. Special consideration is given to the impact Ada has on the existing methodologies and data processing environments within an organization.

FUNCTIONAL SYSTEM SPECIFICATION

To address the needs for trained analysts, PACAGE has experienced instructors capable of presenting classes in various structured analysis techniques. The use of established methods such as Yourdon/Constantine and Gane & Sarson serve as effective vehicles to document functional specifications leading into PAMELA 2 design. These methods are primary among PACAGE's capabilities in education.

SYSTEM DESIGN & PROGRAMMING

PAMELA 2 is the design methodology that stands at the center of the PACAGE development organization. A key factor in understanding the nature and value of Ada lies in recognizing the close relationship that exists between the Ada language and designs for use of the language. Directly stated, an understanding of both programming and design is required to do either effectively. Faced with this dilemma an organization must not only acquire Ada programming knowledge, but also acquire design skills in order to maximize the use of Ada in programming. In turn the programmer must learn to read design documentation in order to implement the design in Ada. Initially a difficult problem, the solution offered by PACAGE is a systematic and coordinated training program consisting of three phases.

PHASE 1 - Taking advantage of the Ada training expertise of AETECH, programmers are introduced to the language and become familiar with its use. The training course includes Ada concepts as well as hands-on use of the language. Automated training and support tools developed by AETECH are used in support of the training curriculum. As these programmers develop, the client may designate those who will proceed on to become software designers. At this time the training program progresses into the next phase.

PHASE 2 - In this phase all trained programmers attend a three-day course on the PAMELA 2 design methodology presented by Thought**Tools, Inc. This course presents in detail the design steps, documentation techniques, and design principles associated with PAMELA 2. Drawing on the knowledge of Ada gained from previous training, the course uses a case study to walk students through the life cycle steps. This course can also be tailored to use case studies more familiar to the client organization.

PHASE 3 - The third and final phase of training available is an independent practical application exercise. In this project those individuals designated as designers are given a case study design project to exercise the skills learned. This case study can also include training on the use of appropriate design support tools. Following the development of the practice design, the designers can work with the programmers to actually implement the design in Ada code.

PART II SUMMARY

The most rapid and maximum benefits of the PACAGE concept are derived through a combination of its capabilities as both a system development and a technology transfer mechanism. The matrix attached hereto as Exhibit 1 provides an overview of how these capabilities can be organized to meet those objectives. Adopting this approach, clients can use PACAGE to develop an automated system in Ada, while simultaneously achieving transfer of Ada technology into their organization. The experience and expertise of the PACAGE team become resources to ensure establishment of a sound Ada software engineering foundation. Production of a needed system can proceed using PACAGE while serving as a source of real world experience for client organization personnel.

The training component of PACAGE helps to bridge the initial knowledge gap and complements the technology transfer process. Placing the training function within the production environment accomplishes two significant objectives. First, the students have the opportunity to learn within the context of a real world system development framework. This has the effect of maximizing the interest of the student and the value of the training presented. Second, training can be immediately followed with the application of learned skills to the project. As part of the development team students continue their education by working with experts as they progress through the life cycle steps. At the completion of this cycle students will have both learned and used the skills required to conduct Ada software development.

This strategy can be adopted for either a pilot project or a full-scale system development effort. The flexibility and capacity of PACAGE allow for tailoring of its approach to best meet the specific needs of the client. It is through the use of this integrated concept that PACAGE is best able to provide total support to DoD and other users in successfully making the transition to Ada.

The availability of key resources to accomplish these critical objectives provides an innovative alternative leading toward realizing the power and value of Ada. With the capabilities of PACAGE available to a client, the decision to implement systems in Ada is both achievable and prudent.

PACAGE CAPABILITIES MATRIX

Ada Life Cycle and Training Requirements	*AETECH*	*NTC*	*Thought **Tools*	*PAC*	*Related Automated Support Tools*
Strategic Planning				√	
Functional Systems Specification				√	AdaGRAPH, Yourdon Toolkit
Training for Functional Analysts and Management				√	AdaGRAPH, Yourdon Toolkit
Systems Design			√	√	AdaGRAPH/PAMELA 2
Design Quality Assurance			√	√	ADAMAT
Training for Designers in Methodology			√		
Training for Designers in Use of Tools				√	AdaGRAPH
Training for Designers and Programmers in Ada	√				Ada Training Environment
Programming	√	√			AETECH/NTC HW/SW Workstation Tools
Software Quality Assurance	√	√			ADAMAT
Acceptance Testing			√	√	
User Training and Fielding/ Installation		√		√	
Maintenance and Enhancements	√	√	√	√	AETECH/NTC HW/SW Workstation Tools

Exhibit 1

REFERENCES

a. Analysis and Design for Ada, EVB Software Engineering, Inc., 1986.

b. Grady Booch, Software Engineering with Ada, 2nd ed., Benjamin/Cummings, 1987.

c. George W. Cherry, The PAMELA 2 Designers Handbook (Preliminary Edition). Thought**Tools, Inc., October 26, 1987.

d. Vance McCarthy, "Ambrose Puts Ada Through the Wringer," Federal Computer Week, December 14, 1987.

This Page Left Blank Intentionally

Ada and the Air Force Institute of Technology

Capt William A. Bralick, Jr.
Capt David A. Umphress

Department of Math and Computer Science (AFIT/ENC)
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433

0. Abstract

Software Engineering is taught at the Air Force Institute of Technology using Ada to express the concepts of "good" programming. This paper describes the introductory software engineering and Ada course at AFIT. The course is presented along with a description of projects implemented during the quarter. Observations on Ada education are also made.

1. Introduction

Even now, Ada means a lot of different things to different people. Some view it as a programming language. Others picture it as a programming philosophy. At the Air Force Institute of Technology, Ada is envisaged as a valuable tool for instructing students in the concepts of computer science. It is used to help students make the connection between theoretical and often abstract principles presented in the classroom with concrete applications representative of the real world.

The Air Force Institute of Technology, or AFIT, is the graduate school of the Air Force. Our mission is to service the professional continuing education and graduate education needs of the Department of Defense. As such, we offer instruction in mathematics, physics, electrical engineering, aeronautical engineering, civil engineering, logistics, and computer science. The last is especially important because intelligent use of computers is gaining increasing visibility in our high-tech Air Force. DoD Directives 3405.2 and 3405.1 have further bolstered support of our computer science program and, indeed, have focused attention on our Ada curriculum.

Ada education is the specialty of the Mathematics and Computer Science Department. With tongue in cheek, we say that we offer courses in Ada. This is only partially true. We offer courses in software engineering, concurrent programming, and environments. Ada is used as a vehicle for expressing the concepts presented in each of the courses.

2. Ada and Software Engineering

The software engineering class, entitled Math 555, is the most popular of the courses and is the main topic of this paper. It is a five-hour lower level graduate course and must be taken by all students receiving master's degrees in computer science, computer engineering, and electrical engineering. It is also required of all Army students participating in a two-quarter non-degree program. Math 555 is offered six times a year; one section is conducted during each of the winter, spring, and fall quarters and three sections are taught during the summer quarter. Approximately 100 to 150 students take the course every year. Texts used are Introduction to Software Engineering with Ada and Software Components with Ada, both by Grady Booch. We supplement the course with MIL-STD-1815A, the Ada Language Reference Manual, and the SofTech Ada Primer written for the Army Communications-Electronics Command. Our students develop software on the Verdix Ada Development System available on the Institute's three Unix computers.

3. Lecture

As mentioned previously, Math 555 focuses on teaching software engineering. As such, the lectures concentrate on teaching problem solving as opposed to teaching language syntax. While a minimal amount of syntax is presented during lectures to illustrate "good" programming practices, the lectures are intended to teach concepts; homework assignments, group projects, and individual projects help students learn syntax. The quarter is divided into six distinct lecture phases:

1.	Software Engineering Principles	(Weeks 1-2)
2.	Design Methodologies	(Weeks 2-4)
3.	Coding Methods	(Weeks 4-6)
4.	Advanced Ada	(Weeks 7-8)
5.	Software Reusability	(Weeks 8-10)
6.	Trends	(Week 10)

Each phase builds on the concepts and principles introduced in the previous phase. The first phase examines the need for good software engineering and the role Ada plays in that effort. The rationale and purpose of Ada are explained. The structure of the next two phases mirrors the traditional software life cycle. Phase 2 examines requirements and design. Phase 3 focuses on coding, testing, and maintenance. Thus, Ada is examined first from the top-down and then from the bottom-up. With this foundation, advanced topics such as tasking, real-time support, and programming environments are introduced. Finally, common algorithms and data structures are presented within the context of reusable software components. This phase is taught primarily at a high level since students do not have enough time to apply the ideas. It, however, introduces them to topics that are covered in depth in follow-on courses. The final phase is an overview of the state-of-the-art of Ada in the software engineering world.

3.1 Phase 1: Software Engineering Principles

The purpose of this portion of the course is to present Ada from the top down. The lecture introduces students to the concept of software engineering and its impact on new language development. A description of the problem of software construction is presented, followed by Ada's attempt to address the problem.

First, the students are reminded of the role of computers in problem solving. This allows them to understand that tremendous advantages may be realized by using computational resources, but that problems occur in effectively representing the problem to be solved. The lecture then turns to a discussion of the gap that exists in the way that humans represent ideas and the way that machines represent information. Using this as a foundation, the "software crisis" is described. The students are then introduced to attempts to structure programming efforts through the use of software engineering. This is done by presenting the concepts, principles, models, methodologies, tools, and environments of effective problem solving. For the most part, the discussion is patterned after McKay [1986]; however, his ideas have been adapted to fit the particular audience. The net result of the lecture to this point is to help students realize that the problems inherent in software development are complexity management issues. It encompasses both the technical and managerial world. To date, however, languages and software engineering have progressed along parallel but separate paths. The lecture then shows that Ada forces the paths to converge into an integrated whole by observing that Ada embraces a software development philosophy by providing not only a single unified language, but also methodology, tool, and environment support.

I. Software crisis -- the problem
   A. Role of computers in problem solving
   B. Information representation gap between man and machine
   C. Software crisis described
II. Software crisis -- the solution path

- A. Software Engineering defined
- B. Accomplished through
  - 1. Concepts
    - a. Modularity
    - b. Top down design
    - c. Structured design
  - 2. Principles
    - a. Localization
    - b. Information Hiding
    - c. Abstraction
    - d. Completeness
    - e. Confirmability
    - f. Consistency
  - 3. Models
    - a. Language
    - b. Documentation
  - 4. Methodologies
    - a. Functional decomposition
    - b. Data flow
    - c. Data structure
    - d. Object-oriented design
  - 5. Tools
  - 6. Environments

III. Software Crisis -- the Ada solution?

- A. Historical perspective
- B. Design philosophy
  - 1. Concepts
  - 2. Principles
  - 3. Models
  - 4. Methodologies
  - 5. Tools
  - 6. Environments

3.2 Phase 2: Design Methodologies

From the first phase of the course, the students have a firm understanding of the role software engineering plays in the problem solving process. This phase of the lecture introduces the students to the principle that software has an identifiable life cycle which must be accomodated by effective planning. A description of the traditional "waterfall" and the prototype software development life cycles are presented along with their corresponding Air Force interpretations. The remainder of this portion of the lecture then turns to discussion of the requirement, specification, and design aspects of the life cycle. Requirements analysis and specification are presented only in terms of how they may be supported by Ada tools and environments. More detail is given to software design. First the students are presented with examples of "Ada-TRAN" as well as what we consider well-designed Ada systems. This serves as a lead-in to object-oriented design and how it contributes to well-engineered software.

Object-oriented design is presented from the standpoint of EVB [1985] and Booch [1983, 1987]. We discuss defining the problem, formulating an informal strategy, and formalizing the strategy by identifying the objects, identifying operations, and establishing visibility between objects. During this time we define abstract data types and abstract state machines and describe their roles in the software development process. This part of the lecture is fairly general and requires very little knowledge of Ada. Following the presentation of visibility, the discussion of object-oriented design is suspended and the students are given their first formal exposure to Ada structure: constructs that support abstraction (packages, subprograms, and, briefly, tasks and generics), lexical elements, and data typing. Object-oriented design is then picked up again by defining how to establish interfaces between objects and how to implement each object. Since the students have not yet been taught about Ada control statements, objects are implemented in an Ada PDL format.

The sequence of topics is intended to ease the students into Ada coding practices, rather than to permit them to develop bad programming habits by coding Ada without a proper foundation in software engineering.

I. Software Life Cycle
   A. Definition of life cycle
   B. Models
      1. Waterfall model
      2. Prototype model
      3. USAF software model
   C. Managerial issues: team organization
II. Requirements Analysis
   A. Definition
   B. How supported by Ada
III. Design
   A. Definition and goals
      1. Examples of good and bad designs
      2. Abstract data types and state machines
      3. Objects and operations
   B. Preliminary design
      1. Define the problem
      2. Develop and informal strategy
      3. Formalize the strategy
         a. Identify the objects
         b. Identify the operations

c. Establish visibility
C. Detailed design
1. Ada features
a. Program structure
b. Lexical elements
c. Typing
2. Establish the interface of each object
3. Implement each object in Ada PDL
4. Recursively apply the process

3.3 Phase 3: Ada Overview

This phase of the lecture builds on those foundations established earlier by introducing the student to the Ada constructs that facilitate good engineering practices. Discussion points are organized topically. In general, the most fundamental and least complicated topics are presented first. The lecture begins with a description of control statements. From there, the students are presented with exceptions, generics and input/output. Each of these constructs have been introduced previously. Here, though, they are presented in sufficient detail to allow the student to comprehend and, more importantly, apply them in programming assignments. Examples are presented during the lecture to illustrate function and form.

I. Control Statements
  A. Sequential
  B. Conditional
  C. Iterative
II. Exceptions
  A. Purpose
  B. Declarations
    1. Predefined
    2. User defined
  C. Exception handlers
  D. Raising exceptions
  E. Propagation
III. Generics
  A. Purpose
  B. Declarations
  C. Instantiation
  D. Formal parameters
IV. I/O
  A. Ada I/O model
  B. TEXT_IO
  C. I/O exceptions
  D. SEQUENTIAL_IO
  E. DIRECT_IO

3.4 Phase 4: Advanced Ada

By this point in the quarter, the students have a good grasp of the features of Ada that are also present in more traditional languages. This phase of the course presents features of Ada that go beyond FORTRAN and COBOL in representing the problem to be solved. Tasking, embedded system support, and programming environments are presented here.

The tasking portion of the lecture begins with a presentation of parallelism. Synchronization, deadlock, mutual exclusion, and starvation are discussed at a level sufficient for the students to understand the complexities of concurrent programming. Ada tasking is illustrated with examples of asynchronous and synchronous tasks, stages of rendezvous, and use of accept and select statements. Anonymous and named task types are presented, as are tasking priorities, multiple entry points, and exceptions. The advantages and disadvantages of Ada for real-time applications are also discussed.

Next, Ada features which support embedded architectures are presented. This entails explaining the form and use of constructs found in chapter 13 of the LRM: address clauses, length clauses, enumeration representation clauses, and record representation clauses.

Finally, program support environments are briefly discussed. Here, we define what a support environment is, what it should do, what tools could possibly be in it, and what characteristics it should have. This usually invokes a spirited debate among the students. At this point in the quarter, they have almost finished the group project and have usually spent inordinate amounts of time working with a montage of Unix tools that are not particularly integrated. By now, they have experienced the woes of software development and become excited at the prospect of environments to ease their administrative development burdens.

- I. Concurrency
  - A. Definition
  - B. Issues
    - 1. Synchronization
    - 2. Deadlock
    - 3. Mutual exclusion
    - 4. Starvation
  - C. Ada tasking

1. Tasking model
2. Asynchronous tasks
3. Synchronous tasks
   a. Rendezvous
   b. Select & accept statements
   c. Variations on select and accept statements
4. Anonymous vs named task types
5. Task priorities
6. Exceptions

D. Real-time issues

II. Embedded system support
   A. Special needs of embedded systems
   B. Language constructs

III. Programming environments
   A. Definition
   B. Uses
   C. Characteristics
   D. Minimal support tools
   E. Interfaces

3.5 Phase 5: Reusability

Ada is presented from a "programming-in-the-large" philosophy. The software factory concept is discussed along with the advantages and disadvantages which accrue from it. The concepts of functional, process, and operational abstraction are illustrated with examples of real-world Ada projects. The majority of the time, though, is spent on reviewing classical data structures and algorithms and how they fit into reusability doctrine. The intent of this portion of the course is to provide the students with a grab-bag of abstractions that will help them conceptualize and implement solutions to problems. As such, reusable components are presented from a "what" vs "how" standpoint. This phase is typically a "cool-down" portion of the course in that it is conducted during the last two weeks, a time in which major new programming assignments are not feasible.

I. Reusability
   A. Software factory concept
   B. Classification of reusable parts
      1. functional
      2. process
      3. operational
   C. Analysis metrics

II. Data structures
   A. Composition fundamentals
   B. Monolithic structures
      1. Stack
      2. Queue

            3. Map
            4. Set and bag
         C. Polylithic structures
            1. List
            2. Tree
            3. Graph
      III. Algorithms
         A. Search
         B. Sort

3.6 Phase 6: Trends

This last portion of the course is the point in which we polish our crystal ball and try to give the students a feel of what the future holds for software engineering, in general, and Ada, in particular. This portion of the class is the shortest, typically one to two days in length, and is included to give the students a sense of closure so that what they have learned may be put into the context of future jobs.

4. Assignments

Student participation is vital to the success of any course. We feel very strongly that this is especially so for programming courses. No matter how effective or well-organized our lectures are, the students still need to develop software. To structure their learning process, we ask them to work homework problems, review technical papers, develop individual projects, and develop software in small groups.

4.1 Group Project

The objectives of the group project are to have the students demonstrate proficiency in designing and developing Ada programs, to apply the ideas of software engineering to Ada, and to experience the dynamics of working in groups.

At the beginning of each quarter, we divide the class into four-five person groups. Each group is charged with the task of selecting a simulation of a closed-feedback loop weapons system, i.e., a video game, that will be implemented by the end of the quarter. The groups use object-oriented design to write a complete design specification of the game and to partition the development task among members of the group. The design is presented to the class in a 25-minute informal critical design review (CDR). Following this review, the groups implement the proposed system using to the maximum extent possible the principles of software engineering discussed in class. A live demonstration of the software is presented to the class in a 25-minute development, test, and evaluation (DT&E) briefing.

Games that been implemented in the past include "Mastermind," "Battleship," "Adventure," "Hangman," and "Lunar Lander." By necessity, the games do not possess large degrees of graphics sophistication. But, they provide a creative outlet for each group.

4.1.1 Critical Design Review. At the critical design review, each group is asked to present

1. Problem statement
2. Brief functional description of what the system will do
4. Assumptions made in the design (e.g., hardware, level of expertise of the user, etc.).
4. Scope of the design
5. Objects and operations
6. Diagrams showing relations among objects
7. Milestone chart

The CDR class period is divided into two parts. First, one group presents their design. Next, two other groups (designated before class time) debate the merits of the design. One group takes the "pro" position; the other group takes the "con" position.

The briefings are informal in format. The design is graded on

1. Oral grade
   a. Clarity of the presentation
   b. Comprehensiveness of the presentation
   c. Linkage to course material
   d. Quality of responses to questions
?. Quality of the documentation
   a. Specifications document
   b. Design document
   c. User's manual
   d. Test plan

We critique the design independently of the debating group and make recommendations for improving the design.

4.1.2 Development Test and Evaluation. The DT&E presentation is intended to discuss design changes since the CDR class period and also to demonstrate the software. The presentations include

1. Problem statement
2. Design diagrams of the current implementation
3. Live demonstration of the software using a large screen projector

The briefings are informal in format; grades are assigned on the oral presentation based on the same criteria as the CDR. Written documentation must include amendments to each of the CDR written deliverables as well as the source code for the game.

4.1.3 Operation Test and Evaluation. We ask each group to make a modification to the implementation as presented at the DT&E. The change is used to evaluate the robustness of the design in an pseudo-operational test and evaluation (OT&E) setting. Each member of the group individually implements the requested modification.

The OT&E portion of the project is often as challenging for us as it is for the students. We specifically look for design flaws and ask for modifications that illustrate those flaws. The students are evaluated on their ability to integrate the requested change into the overall design, as opposed to "band-aiding" the change. From the student's standpoint, this is an exercise in code maintenance. Further, students that have fully participated in the project and are familiar with the implementation have a better chance of successfully making the change than do students who have chosen not to participate. The OT&E assignment often generates the commitment to full participation during all phases of the project development.

DT&E is held the eighth week of the quarter; OT&E modifications are due at the end of the course on the tenth week. Each student is required to turn in a copy of her or his source code (with modifications highlighted), any amendments to the CDR or DT&E deliverables, and a short report on managerial and technical lessons learned.

4.2 Individual Project

The individual project assigned is to implement an abstract state machine: Rubik's cube. The cube was chosen because of its familiarity, structural regularity, and relative simplicity. Most students have at least seen one at some point in their lives and are able to decompose it into a set of objects and operations. Fu-ther, most students are able to solve the problem partially because the cube is a concrete object with relatively few levels of abstraction. Even so, for students who have had some practice in analyzing objects, the Rubik's cube proves challenging. It should be noted that the problem is not to implement any of the several known solution algorithms for the cube, but to merely provide an encapsulated representation of the cube which could then be manipulated either interactively or by a test driver.

The objective of the individual project is to hone design and implementation skills needed for the group project. The design of the cube is due before the design of the group project; similarly, the implementation of the cube is due before the implementation of the group project. The cube is instructive in that it naturally decomposes in an object-oriented fashion. Moreover, the representation of the cube takes the form of an abstract state machine as opposed to the more familiar abstract data type. Thus, the problem is simplified somewhat by requiring that no objects be exported; instead, the object must be created, maintained, and manipulated within the body of the package. As expected, this subtle point eludes many students and therefore is useful in explaining the difference between abstract state machines and abstract data types.

4.3 Technical Paper Reviews

During the course of the quarter, we ask each student to read, summarize, and evaluate two recent published articles. The source and choice of articles are up to the student; the only condition is that they have to relate to Ada is some way. The objective of this exercise is to familiarize the students with information sources on Ada as well as increase awareness of the role Ada plays in the computing community. It also gives them an opportunity to research the application of Ada in a specific area of interest.

The reviews are two to five pages in length and include

1. Full bibliographic citation
2. Summary of the article
3. Assessment of how the information presented in the article affects the DoD and/or them.

While some students find this unnecessary busy-work, many students are surprised and impressed by the volume of Ada literature.

5. Observations

Overall, the course is received favorably by the students who take it. Points worthy of note, though, include the workload, textbook, Ada syntax, object-oriented design, and projects.

5.1 Workload

Obviously, the course entails a lot of work. However, we feel that for five graduate hours, the work is not excessive. We are in the classroom one hour for four days a week and the laboratory for three hours once a week. Lectures, labs, and assignments are coordinated so that nothing is expected of the students unless it has first been discussed in class. The students initially express concern over the amount of work, but soon realize that they can not absorb the material unless they get "hands-on" experience.

5.2 Textbook

The primary text that we use has been the source of many problems. We require the students to buy a copy of Booch's book [Booch 87]. While the text provides good examples of solved design problems, it is inadequate for teaching useful Ada. Examples are not complete and are very easy to take out of context. The narrative is obscure and often full of unnecessary "Ada-ese." We also feel that the book does not address the concepts of software engineering to the level needed by the students.

We supplement the text with other readings, each of which is weak in one or more areas. The combination of material, however, provides a good Ada background. The LRM has played an extremely important role in providing comprehensive -- and complete -- examples of Ada.

5.3 Software Engineering vs Ada

Often, students come into the classroom expecting a course on Ada syntax. They are surprised when we spend much of the first lecture phase on software engineering. Indeed, postponing "serious" discussion of Ada for 15% of the course seems frustrating to many; however, they see the value of it by the end of the course. We are careful, though, to ensure that we do not dwell on software engineering too long: the "why aren't I coding syndrome" can only be staved off for a short while. We find that talking about software engineering concepts during the first week is palatable if a trivial Ada compilation exercise is given as a homework assignment. For the most part, the end-of-course critiques thank us for teaching problem solving and not just another programming language.

5.4 Group Project

The group project is the most time consuming of the assignments made in class. But, it has the most impact on teaching object-oriented design. The interaction of four or five people working on determining how to represent the project as objects and operations fuels thought processes and gives an appreciation of the design procedure.

We have found that the object-oriented design process of Booch's first edition [1983] works better than that described in his second edition [Booch 1987a]. The 1983 edition describes a procedural process that entails writing a one paragraph informal strategy of the problem to be solved. The objects and operations are then determined by parsing the paragraph. The 1987 edition describes object-oriented development, a method which derives objects and operations through intuition and knowledge of the problem. Groups that follow the 1983 method produce designs that are more consistent and uniform than groups that follow the 1987 edition. Students indicate that having to write a paragraph describing what must take place forces them to visual the problem. They are then able to identify objects of interest (whether they formally underline nouns and verbs or not). If they simply write down what they think are objects without using the informal strategy approach, they loose sight of hierarchically organized software and produce modules that are not at one consistent level of abstraction.

5.5 Individual Project

As a pedagogical tool, Rubik's cube (or similar mathematical puzzles) excels for several reasons. First, the cube can be held in the student's hand and manipulated thereby promoting interaction between modeler and model. Second, a good engineering solution is a fairly obvious one, a situation which builds the student's confidence. And third, the hidden requirements (e.g., the cube has to be tested therefore some kind of additional package has to be built) just like in a real life engineering situations can catch the unwary, especially if a reasonable development schedule is not maintained.

The potential exists for some truly elegant representations, however, none have been produced to date. The reason is probably that there is nothing wrong with the most obvious solution: arrays. Arguably, since the array-based representation is so obvious, a tricker representation would be worse from a software engineering standpoint since it would be less understandable.

A suggested variation of this assignment, perhaps one to challenge advanced students, would be to require that the solution be implemented without resorting to arrays. Or, alternately, require that the student base her or his solution on a given reusable software component such as found in Booch [1987b]. If approached reasonably, this could provide the student the additional benefit of learning to correctly use a given reusable module, and learning to operate within the engineering constraints imposed by reusability. Another alternative would have the students each assigned a different reusable component with which to implement the cube.

5.6 Student Expertise and Ada

The demographics of students that take Math 555 is an oddity. The majority of our students are either seasoned programmers or officers with very little computer experience. This has provided us an interesting insight into the way we teach the course. The inexperienced officers are overwhelmed with Ada. They see it as the legendary 200-blade Swiss Army knife: complicated, difficult to learn, and quite formidable. The experience programmers cavalierly see Ada as simply another programming language, one to be approached much the same as Pascal, BASIC, or FORTRAN. The novice programmers' biggest problem is overcoming their fear of the compiler; the experienced programmers' biggest problem is ignoring the software engineering strengths of Ada and writing Ada as Pascal. Interestingly, though, the novice programmers are usually able to grasp the concepts of object-oriented design. To them, the method is very natural and Ada is then a good way of representing solutions to problems. The experienced programers, on the other hand, have a difficult time comprehending the object-oriented design mindset. They are intent on using functional decomposition and often perform poorly on the initial design homework.

We counter these problems by forming groups for the group project the first week of the quarter. We assign experienced and inexperienced people to the same group. The camaraderie (of despair, often) that usually develops benefits both types of people. Not only do they work on the projects together, but also work on homework as well. The "carrot" of having the OT&E modification pending at the end of the quarter normally gives all group members sufficient impetus to participate as closely together as possible.

6. Other Ada-Related Courses

As mentioned earlier, AFIT offers a course in concurrent programming and one in software environments. As of this writing, the concurrent programming course, Math 655, has not been taught (but will be held in Spring 1988). Its objectives are to first, instruct students in parallel programming techniques, and second, teach them how to use the constructs of Ada that support concurrency. We plan to spend the majority of the quarter discussing design of concurrent systems, pitfalls of concurrency, synchronization techniques, analysis of parallel algorithms, and implementation of real-time systems. We also plan to have the students work on a large software project having to do with a concurrent application.

Math 755, the programming environment course, has been taught twice in the past three years. It is a doctoral level seminar course that gives students a chance to read papers on the state-of-the-art in environments. Although the primary emphasis is on Stoneman and the Ada Programming Support Environment, students are exposed to a number of different conceptual environments. Future course offerings will include an in-depth investigation of the Common APSE Interface Set (CAIS).

7. Summary

Ada at the Air Force Institute of Technology is flourishing. We believe that our software engineering course helps the students who take it to understand the complexities of software development. We feel strongly, too, that there is no such thing as an Ada programmer. We educate Ada problem solvers. Gone are the days of teaching program language syntax courses. We strive to nurture good software development habits and feel our efforts are amply supported by Ada.

BIBLIOGRAPHY

Booch, Grady. 1983. Software Engineering with Ada. 1st Ed. Benjamin/Cummings, Menlo Park.

Booch, Grady. 1987a. Software Engineering with Ada. 2nd Ed. Benjamin/Cummings, Menlo Park.

Booch, Grady. 1987b. Software Components with Ada. Benjamin/Cummings, Meno Park.

EVB. 1985. An Object-Oriented Design Handbook for Ada Software. EVB Software Engineering, Inc.

McKay, Charles. 1986. Presentation at the First Annual ASEET Symposium. (Orlando, FL, June 1986).

This Page Left Blank Intentionally

Incorporating the Use of Design Methods into an Ada Laboratory Curriculum

Karyl A. Adams
c.j. kemp systems, inc.
Huber Heights, Ohio 45424

As the Ada curriculum at the Air Force Institute of Technology (AFIT) has matured, the structure of the supporting laboratory has undergone significant changes. Every effort has been taken to ensure that the practical applications which the students develop follow a disciplined method. To support this goal, formal use of the Object Oriented Design method coupled with design reviews of project work were incorporated into the structure of the lab work. This paper shall document how these improvements have been added and what the impact has been on the students and the work they produce.

Specifically, this paper shall document the maturation of the lab curriculum over the past couple of years. In the process, the paper shall present the changes made to the way in which project work was assigned and directed, the student reactions to the labs, and conclusions regarding the merits of the lab work. This is a qualitative study, with conclusions based on discussions with students and other faculty members, and observations of the course instructor. However, such a study may assist other Ada educators as they try to structure a manageable, educational Ada lab program.

The paper describes a case study of the laboratory exercises and structure used with two recent offerings of the introductory Ada course at AFIT. While it represents a single data point and one instructor's approach to incorporate more software design, it is also representative of the changing way in which Ada software engineering labs are being run at AFIT. In the paper, the projects and exercises assigned for the two labs are described. A comparison is then made between the two classes of graduate engineering students taught by the same instructor. This comparison evaluates the effectiveness of the two lab approaches by discussing student results.

BACKGROUND

With the more traditional methods of teaching computer languages, students are taught the mechanics of using the language with a variety of programming assignments. Such assignments typically start with the fundamental pieces of language syntax and increase in complexity until the student has been exposed to the majority of the language's syntactic features. Such assignments focus on the mechanics of using the language, the associated compilation system and its proper use, and the production of correct results. Success is measured in these terms. Infrequently has the emphasis been placed on the design phase of software development and the formal incorporation of software engineering techniques and methods. Even with today's knowledge of software engineering principles, many language courses still feature companion laboratory work focused on syntax-based exercises.

The addition of Ada into the mainstream of a computer science and engineering curriculum forces a review of the methods used in the companion laboratory. At AFIT, the focus in the introductory Ada course has been on software engineering. This is reflected even in the title of the course, "Introductory Software Engineering with Ada". The lecture portion of the course has matured in the past five years into a strong introduction to the concepts as discussed in current software engineering literature [Bralick and Umphress, 1988].

It has been somewhat more difficult to redefine and place into operation a supporting lab program which addresses the practical use of Ada in an appropriate fashion. In part this difficulty is one of attempting to depart from a familiar style of instructing. Additionally, there are few guidelines which help break the pattern of training in syntax use and support development of a program with a clear, strong software engineering emphasis. Certainly students must learn to use the language features correctly, however the contention is that correct use of syntax is but a portion of the Ada educational process.

As AFIT's Ada expertise has increased, the character of the supporting lab program has changed dramatically. In the earliest offerings of the Ada courses, the lab was little more than a set of exercises to look at the syntax of the language. As the knowledge of the language and software engineering grew, the lab was changed to reflect this new appreciation for how Ada could and should be taught [Lawlis and Adams, 1987].

Of particular interest in this paper are two recent offerings of MATH555, "Introductory Software Engineering with Ada" which were conducted during the summer quarters in 1986 and 1987. Each course was taught to a group of masters candidates by this author. The course lecture material remained largely unchanged from one offering to the other. The laboratory material was altered by placing an even greater emphasis on formal design and review. The lab programs and the results from the two sections are discussed and analyzed in the following sections.

THE NATURE OF THE STUDENT POPULATION

It would be misleading and incorrect to assume that the two classes in question were exactly "equal" to one another in their ability, motivation, and desire to learn about Ada and software engineering. However in general, the personalities of the MATH555 sections have had several key features in common. Since the class makeup is so similar, it seems acceptable, from a qualitative standpoint, to compare and contrast their progress.

The nature of the classes will be described in order to characterize them better. First, all the students in the two sections of interest were candidates for masters degrees in either computer engineering or computer science. Thus each student had a technical undergraduate degree. With the exception of one student who was a civilian employee of the Air Force, all the class members were military officers.

Each of the sections was characterized by a diversity of backgrounds and experience. Each class had a handful of students with extensive backgrounds in programming as well as a far greater number who had little or no experience. With rare exception, none of the students had received training in software engineering prior to this course. Each class also had students who had recently completed undergraduate training, as well as students who had been out of school for several years. Each of these affected how individual students accepted and performed in the lab. These effects will be discussed later in the paper.

Each of the sections received the same number of lecture and scheduled laboratory hours per quarter. The course consists of forty lecture hours, held one hour a day, four days per week for the duration of the ten week quarter. There are thirty lab hours, with one three-hour lab per week.

So far, the characteristics discussed illustrate the likenesses between the two sections. Certainly for comparison purposes the more the classes were alike, the better. There was one distinct difference however. The class taught in 1986 contained over thirty students while the 1987 class was much smaller with only seventeen students. There is little doubt that the smaller class size, as well the instructor's added experience of teaching the course another time, was an advantage for the second class. It is difficult, if not impossible, to ascertain how significant an impact this had. As will be discussed later, it appears from the students comments that other factors outweighed this one.

ORGANIZATION OF THE LABORATORY CURRICULUM

In this section the laboratory assignments for each of the two Ada sections is defined. The reader should quickly recognize that there was more formality in the lab process for the second course offering. The addition of formal reviews to determine student progress was the critical step in creating a more realistic, formal use of the software engineering concepts being discussed in the class lecture.

DESCRIPTION FOR THE FIRST SECTION - 1986

For the 1986 Ada class, the laboratory assignments consisted of a set of isolated programming problems, an individual project, and a group project. While the student certainly learned things from each assignment that could prove useful in later efforts, there was no attempt to relate the three activities.

The small programming exercises did focus on the rudimentary features of the Ada language in order to assist the student in writing syntactically correct Ada code. The Telequiz system that can be purchased as part of the TeleSoft compilation system was used for these problems. Each Telequiz focused on a specific piece of the Ada language (for example, array representation or use of loops) and gently introduced the student to the use of Ada to represent a solution.

In accomplishing the Telequiz work, students were not required to use a formal design method. They were encouraged to use the techniques being discussed in class. The problems to be solved were so small and self-contained, and their solutions so apparent, that requiring the use of a design method was deemed inappropriate.

The Telequiz effort made up 10 percent of the students course grade. Each Telequiz was graded based on the following criteria:

Design (40%)

- Use of meaningful, helpful comments in-line with the code, reflecting the students design strategy by describing algorithms and data structures
- Use of argument passing for variable control
- Use of appropriate data structures

Style (25%)

- Use of proper control structures
- Use of modularity
- Use of indentation
- Use of meaningful names

Execution (35%)

- Compilation without error
- Execution to normal termination
- Results correct according to exercise specification
- Ease of understanding
- Ease of use

Over the course of five weeks, the students were assigned eleven of the basic Telequizzes. As each was so short, several could be accomplished in the three hour lab session available each week. Concurrently with the Telequiz effort, students were working on their individual projects.

The individual projects for the first class required the design, implementation, and testing of an Ada package for a Binary Search Tree (BST). The package was required to "protect" the integrity of the underlying BST structure. It was to export the BST type as well as operations to create and maintain a single tree, add and delete individual tree elements, determine the existence of a given value in the tree, and traverse the tree in preorder, inorder, and postorder fashion. It was not required that the students approach their design from the standpoint of component reuse, thus a generic package was not mandated.

The individual project (which was worth 20% of the course grade) was graded based on the strength of the design (40%), the resulting implementation (40%), and answers to a set of questiong regarding how the design could be extended to handle different situations (20%). For the design the students were required to use the Object Oriented Design method as discussed in class lecture and as defined in Booch and EVB [Booch, 1983; EVB, 1985]. At no time during the development was the design formally reviewed and critiqued, although students were encouraged to seek assistance from the instructor if they had any questions or difficulties.

The group project for this class was intended to illustrate the advantages and disadvantages of working in a group situation. Additionally, the students were to design a useful tool for the research Ada Programming Support Environment (APSE) at AFIT. Most of the groups elected to design generic packages for advanced abstract data types. One group elected to research the available Ada resources at AFIT and complete a design which would provide an advanced "help" facility in the AFIT APSE. Another group designed, developed, and tested an Ada pragma interface capability which provided access to a proven set of graphics routines for the APSE.

All projects were required to use OOD in order to complete the design phase of the work. Students were graded based on the design effort, style and correctness of the resulting implementation, and the ease with which the tool could be included in the AFIT APSE. The group project grade was worth 20% of the overall course grade.

DESCRIPTION FOR THE SECOND SECTION - 1987

Based on a reassessment of the lab program from 1986 and previous years, the 1987 supporting lab assumed a new look. The basic structure of simple exercises, individual project, and group project remained intact, but the nature of those assignments was quite different.

The simple, Telequiz-based assignments, became a much smaller part of the work effort. The number of exercises was reduced to a set of four quizzes which supported the basic introduction to Ada use. Augmenting the quiz work was the addition of demonstration programs to illustrate Ada packaging techniques and tasks. These demonstration programs consisted of well defined, small systems, which the student could see run, make controlled changes, and observe the effects. All of this could be accomplished without detailed Ada knowledge and yet served to introduce the student to the language concepts.

It was in the area of the individual and group projects that the second lab structure differed most significantly from the first. In this offering of the class, these two projects were closely related to one another. The product of the individual effort fed directly into the group effort.

The individual project was a design effort only. Each student was to analyze a set of provided system requirements and then use Object Oriented Design to define the preliminary design for the system. Once the individual projects were completed and had been graded, a set of promising (but fundamentally different) designs were selected by the instructor for further development. Each group was given a selected design for which they were to complete a detailed design and a working implementation. The design given to a group had been the individual work of one person in that group, thus the group had the original designer in case of questions.

The project selected involved a hypothetical Remotely Activated Tracking System (the RATS) which could be used to explore planetary surfaces. The RATS had several elementary functions which included moving around the terrain, recognizing and avoiding obstacles, requesting assistance if necessary, and reporting back to Earth its findings when requested to do so.

The entire project process was couched in a realistic, although controlled, software life cycle set of procedures. The mock Statement of Work was issued by NASA (aka the instructor) and was received by potential bidders (aka the students). There was a design kickoff meeting at which time the bidders could request any necessary clarifications from NASA. Prior to submitting their individual projects for grading, there was a preliminary design review. Bidders were called upon to discuss their designs in an open forum. They were then free to incorporate any last minute changes that were identified in this open review that they felt could strengthen the preliminary design before submission to NASA for full-scale development consideration.

At the time of the preliminary design review, each student submitted their work for the instructor's review. The work was assigned an intermediate grade, and recommendations for any necessary improvements were made. This extra grading permitted the students to see how their work was progressing. It also gave each student the opportunity to improve their understanding of the design method before the final submission of their work. They were then given the opportunity to improve their designs before a final project grade was assigned.

Students were required to submit several deliverable items for this phase of the project. First, each student had to be prepared to present a short briefing at the preliminary design review. For submission of the design itself, they were required to provide their statement of the problem and any underlying assumptions they made, any data which supported their design decisions, a design form which clearly represented the Object Oriented Design method, compiled Ada package specifications for the design, and an indication of the static dependencies among the modules identified.

The preliminary design, which accounted for 20% of the course grade, was graded in two areas - an evaluation based on the stated requirements and an evaluation of the student's performance in meeting those requirements. The first category looked at basic adherence to the stated requirements - were all the pieces provided, did the student apply the design method properly, did the student use Ada properly, and was the report done professionally. The second area was a more subjective evaluation of the student's response to the design and review process. The intent was to evaluate whether or not the student was trying to use the method and learn from the process.

After the preliminary designs were completed, candidates for further development were selected and development teams formed. As part of the group project, students were required to complete the detailed design, implement the design, test their system, and participate in a final design review. Additionally, the group leader was responsible for establishing the schedule for development and determining the priorities for that schedule. In the event that the full system could not be finished (a likely event), each group had to have considered that possibility and defined their strategy for accomplishing the work in a reasonable fashion.

Prior to submitting the work for grading, NASA held a final design review. During this review each group was given one hour to present, demonstrate, and defend their final project designs. Since the designs selected had fundamental differences, the peer evaluations were quite enlightening and lively.

Each group was to submit a final report which contained the preliminary design, any changes to that based on group consensus, the detailed design, all resulting Ada code, the management plan clearly indicating the responsibilities of team members, a test plan for the project, and a self-assessment of the group's activities during the project work.

This discussion presents the two lab approaches in an overview fashion. Prior to analyzing these approaches from both the student and instructor perspectives, it is important to spend a few words in an attempt to show clearly how these two labs related to other MATH555 sections which were being taught during the same time frame, but by other instructors. The next section provides that insight.

COMPARISON WITH OTHER ADA SECTIONS

In 1986, there were two large (30 or more students) MATH555 sections. The lab activities for the other group of students followed a more traditional programming approach in which students were assigned several, relatively large programming exercises during the course of the term. In that section more emphasis was placed on learning the Ada language and less was placed on the use of a formal design method. Students in the other section had many more opportunities to demonstrate their grasp of the language during the course of the several major exercises.

In 1987, there were three MATH555 sections which varied in size from 17 to around 25 students. These classes were more closely coordinated, both in lecture and in lab, than in past years. Although the lab assignments were different, the emphasis on Object Oriented Design and software life cycle issues were emphasized in all three classes. Each class had the same basic structure of individual and group projects, although the specific content varied among sections. Thus there was more commonality among sections in the most recent MATH555 offering.

OBSERVED STUDENT PERFORMANCE

There are several points which became immediately obvious from both of these MATH555 classes. One of the most interesting is that students who enter the class with a great deal of programming experience do not necessarily do well in the class. In fact, when the emphasis is on using a disciplined design method rather than producing code, these students often do far less well than their seemingly less prepared classmates.

In both classes, those who grasped the concepts of Object Oriented Design most rapidly, and most thoroughly, were those who had little, or no, software backgrounds. In fact, one of the strongest students in either section was an Army captain with a business degree who later transferred to AFIT's Graduate School of Systems and Logistics. He had absolutely no preconceived notions of how to develop software. He entered the course with a "clean slate" and had no bad habits to unlearn.

On the other hand, students who had written a lot of software in the past seemed to fight the process to a greater or lesser degree (depending on the student). Their progress was slower and more difficult as they tried to get past the way they had always worked in the past. For the most part this transition was made, but for some it was a difficult process.

A second note of interest was that the more formally the design method was used, the better job the students did in their work. The first class used Grady Booch's first edition of Software Engineering with Ada . The second class used the second edition. In preparing the second edition, Booch relaxes some formality found in his previous edition when presenting the several steps of the design method. The introductory steps found in the first text which address stating the problem and analyzing it are handled in a much more nebulous fashion in the second.

This less rigorous approach causes some problems with beginning students. In was clear early on in all three sections of 1987 that more formality was needed. The method taught during class lecture referred to Booch's early text. Students were told to use the EVB Handbook [EVB, 1985] to direct their design efforts. It was clear during the 1987 class particularly, that those students who took the time to use the handbook had distinctly better designs than those who did not. Students who used the less rigorous approach produced designs with flaws, often serious ones.

Finally, it was also apparent that the overall development process was much more successful during the second class. With the two projects being related, the students didn't spend so much time "gearing up" for the new effort. They worked with one problem specification for the majority of the quarter. The problem was complex enough that it required them to use, or consider using, a majority of the topics covered in class in its solution. It was interesting enough and had enough flexibility in its definition that they didn't lose sight of the challenge. Placing them in a more realistic context encouraged students to enter the role-playing aspect of the development which made the process more fun for them. One of the oft repeated comments from the first offering was that the data structures focus was fairly challenging, but not too interesting.

STUDENT REACTIONS TO THE CURRICULA

In general, the student criticisms and comments from both sections had a lot in common. Almost to a person, students find that using the Object Oriented approach is difficult to grasp,at least initially. Again, almost to a person, they admit that even though they may not have liked the day to day emphasis on design first, their products are better for having had that emphasis.

Student comments also tend to indicate that they do not feel as comfortable with the details of the Ada language as they had hoped. However, they feel that they can define a problem and develop a design for it. It is this instructor's opinion that this is not only an acceptable, but a desirable, reaction to the course. When looking back over the course, the majority of students indicate that the emphasis is good. They indicate an appreciation for having been given a set of tools for "thinking purposes". As long as they can consider their problems in a discipline fashion, the details of the language will come with experience.

The areas in which the comments diverge have to do with the content of the actual lab assignments. Students in the earlier section complained vigorously, and justifiably, that the Telequiz system was over-emphasized. It was useful for the very naive software developer who needed a very simple introduction to syntax, but it was way too much time spent on mundane activities for most students. Thus its emphasis was greatly diminished for the second section.

The first section also did not seem to have as much fun as the second group. While they did good work, the project content didn't offer as much in the way of amusement or reward for a job well done. By involving the second class in a more realistic scenario, with a project that piqued the interest better, those students actually enjoyed themselves more. An observation to be made here, is that they also seemed to work much harder, because the project permitted them the latitude to check alternative design decisions.

The overall comments from the second class indicated that the process used for the lab was successful. The project needs some refining, which came as no surprise, but the focus on the software development process was valuable. While most students had complained about spending "so much time" on the design and "having to wait so long" to start coding, they admitted that the approach helped them produce better project work.

The class reaction in both cases was generally positive. The reaction to the second lab approach was more uniformly liked by the students. Judging from the overall class grades, this reaction could seem surprising as the average grade for the second class was lower than that for the first. The lower grade average was largely a result of the increased scrutiny on the design process.

SUMMARY AND CONCLUSIONS

There are several issues that need to be addressed by way of summarizing the results from this informal study. First, and foremost, it is clear that the students are receiving a better education in the appropriate use of both Ada and the methods of software engineering when the lab is structured around those concepts. The student comments reflect an appreciation for this fact and the Ada code that results from this approach is better defined. It is equally clear that establishing and using a lab with such an emphasis is a difficult task.

From the comments and reactions of the two sections discussed in this paper, it is apparent that the second approach was more effective. Students were more highly motivated with the RATS project. In fact, the more the groups became involved in adopting the role of a contract bidder, the more enjoyable and realistic the effort became. One group, which produced an outstanding project, formed their own "company" for the purposes of the project. The entire design process was tracked via internal memos and reports. The project effort was produced smoothly, with superior supporting documentation, and clearly indicated the group's understanding of both the design process and the use of Ada. They completed a fine design and development effort and managed to have some fun in the process. Introducing a hint of fun now and then into the educational process can be rewarding and valuable for student and instructor alike.

A related comment concerns the formality of the Object Oriented Design method itself. As discussed earlier, Booch has removed some of the formalism in his second text. This author feels that to do so is to weaken the method. The object oriented approach, as any other design method, needs the formality to be successful. The best solution found to date has been to have a student study the handbook prepared by EVB which describes the method in detail, and then apply the method in the form illustrated by EVB. Student designs using this approach have been more complete and robust as a general rule.

A major problem that is always encountered in classes such as these has to do with the skilled "programmer" who wishes to conduct business as always. These folks can be difficult to reach. It is fundamentally difficult to unlearn acquired techniques even if they can be shown to have less chance for success than newer techniques. The student with lesser programming skills almost always embraces the software engineering approach more comfortably and quickly. The key seems to be in clearly contrasting the differences between programming and software engineering, identifying the known benefits of the latter. There will be some who refuse to embrace the newer techniques. They will be outnumbered by those who do try to understand and apply the methods.

A final comment, which has less to do with actual curriculum than it does with the management of it, regarding the teaching of the MATH555 class. One tremendous benefit realized in the second offering had to do with the coordination among the several course instructors. The three instructors that had sections in 1987 shared a common office area which made coordination of the classes and labs very easy. It was commonplace for the group to share experiences, evaluate from day to day the progress of not only their students, but themselves. This on-going analysis and comparison of approaches that did and didn't work proved invaluable in fine tuning the presentation of materials to the classes. It is a scenario which has much to offer both to the instructors as well as their students.

In conclusion, AFIT's Ada program has seen tremendous growth over the past years. The laboratory program finally seems to be maturing to match the quality of the lecture which it is to support. Couching the lab work in as realistic a setting as possible, within the time and resource constraints of an academic environment, supports higher levels of learning. Students are more challenged, they enjoy the work more, and they gain a deeper insight into how the pieces of the Ada puzzle work together to solve representative problems.

BIBLIOGRAPHY

Adams, Karyl A., course materials and assignments for MATH555, Summer Quarter, 1986.

Adams, Karyl A., course materials and assignments for MATH555, Summer Quarter, 1987.

Booch, Grady. Software Components with Ada . Menlo Park, Calif : Benjamin/Cummings, 1987.

Booch, Grady. Software Engineering with Ada, 1st edition . Menlo Park, Calif : Benjamin/Cummings, 1983.

Booch, Grady. Software Engineering with Ada, 2nd edition . Menlo Park, Calif : Benjamin/Cummings, 1987.

Bralick, Capt William A. and Capt David A Umphress, "Ada and the Air Force Institute of Technology", paper presented to the Third Annual ASEET Symposium, June 1988.

Department of Defense, Ada Programming Language ANSI/MIL-STD-1815A, 22 January 1983.

EVB, An OOD Handbook for Ada Software . prepared by EVB Software Engineering, Inc. 1985.

Feldman, Michael B. Data Structures with Ada . Reston, Virginia : Reston Publishing Company, 1985.

Lawlis, Major Patricia K. and Karyl A. Adams, "Introducing Ada and Its Environments into a Graduate Curriculum", paper presented to the Second Annual ASEET Symposium, June 1987.

The Myth of Portability in Ada

Major Charles B. Engle, Jr.
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract: There has been much written about the versatility of Ada and all of the capabilities of the language. In particular, there has been much written and discussed about the capabilities of the Ada programming language to support reusability and its partner, portability. There are those that claim, wrongly, that Ada provides portability by nature of its design. Nothing could be further from the truth! Ada simply provides many facilities and mechanisms which can be used to support portability. The design of the underlying software system provides the portability of the system, not the language in which it is implemented. This paper discusses one particular porting effort and the design decisions which made that effort extremely difficult. It then presents some common rules-of-thumb to consider when designing software systems and implementing them in Ada.

Introduction

The programming language Ada has been standardized since 1983. As the Ada culture has grown up around the language, there have arisen many myths about Ada's capabilities. Some of these myths are apparently the result of deliberate hyperbole used to spread the use and acceptance of Ada, some are the result of ignorance, and some are the result of misunderstandings. It is time that these myths are debunked.

One myth which is often accepted about the programming language Ada is that it is portable from one system to another. Those who spread this myth point to the Ada Compiler Validation Capability (ACVC) as a watchdog system to ensure that all compilers are alike, which, they reason, must surely imply that programs written in Ada can be moved from one compiler to another, or from one system to another, and be successfully compiled. This implication is clearly false, as anyone who has tried to port a large system written in Ada will attest. It is the purpose of this paper to expose the myth of portability in Ada.

First, a simple definition of portability is in order. "Source code portability is a programming language characteristic" which may be defined as the extent to which "source code may be transported from processor to processor and compiler to compiler with little or no modification." [Pressman 82]. Other interpretations of the term are possible. For instance, software portability is defined in [Lecarme 86] as changing "the environment of any given software product to make it work on more than one computer and with more than one operating system." Lecarme and Gart also note that "some authors distinguish 'portability', where no modification is necessary from 'transportability', where a certain effort in adaptation is required. This distinction seems artificial since 100 percent portability is almost never achieved ..." [Lecarme 86].

In some sense it may be more ot a goal than a fact to expect to be able to easily move software from one machine or environment to another. Wiener and Sincovec acknowledge that "total portability of software is difficult to achieve even when using a standard version of a high level language." [Wiener 84]. They attribute this difficulty to "machine dependencies ... in connection with input and output as well as number precision for floating point computation." [Wiener 84]. At least some of the difficulty in portability can also be attributed to character set differences, machine architectures, and file manipulations [Wiener 84].

Given these definitions, it is somewhat naive to believe that portability can be achieved on large software systems written in any language. Rather, what must occur is the maximization of the degree of transportability. This can be facilitated by the choice of language, but not nearly as much as by a well considered system design. Isolation of system dependent features into a clearly labeled package(s) is an indicator of a good design, *i.e.*, one which greatly facilitates efforts to transport software systems.

Therefore, it is a myth that Ada provides portability. Rather, it is the application of sound software engineering principles that can best support efforts to port software systems. Systems must be designed from the very beginning for reuse and portability. Does this mean that software systems written in Ada are less portable than systems written in other languages? On the contrary, Ada provides a rich assortment of mechanisms necessary to facilitate the implementation of software systems which are designed for portability. However, it is not the use of Ada which makes a software system *inherently* portable, but the *design* of the system which Ada implements that makes that system portable. It is not at all difficult to design a software system which cannot be easily ported and implement it in Ada. Similarly, it is entirely possible to design a software system carefully and implement it almost any high level language such that it highly transportable.

This paper will discuss, by way of an example, the Flight Dynamics Analysis System (FDAS) written for the National Aeronautics and Space Administration (NASA). This system is entirely coded in Ada, but it is not cost-effectively transportable. This system was adopted by the Software Engineering Institute (SEI) as an Ada Artifact for educators. It was later rejected for reasons which will be described below. This paper will discuss some of the things that the designers did and some of the design decisions that they made, which inextricably tied their system to a single vendor-specific computer and its associated operating system.

The paper will then discuss some rules of thumb for designing portable and reusable software systems implemented in Ada and explain how and why the features of Ada provide support for these designs.

Porting the FDAS: Background and Problems

Background. The Ada Artifact Project of the SEI grew out of a desire for educators to have a medium to large-scale example of a commercially prepared and used software system to demonstrate software engineering principles and practices to their students. Some of the main gains from software engineering are realized when the systems under development are large. Smaller systems do not often demonstrate the need for the configuration management, reusable code, modular design, etc. It was felt that if educators had access to a larger system, they might be able to involve their students in a learning experience which would demonstrate the need for and use of software engineering. The solution, as conceived and proposed by Dr. John Brackett, then at the Wang Institute, and implemented by the SEI, was to search for an artifact which was sufficiently complex and large to be used for this project. The SEI would then obtain the rights to use and distribute this artifact to affiliates and other interested educators.

After a thorough search of the current market place, several possible candidates were identified. A competitive rating of each candidate was used to select the one which seemed most appropriate for the artifact project. The successful candidate was the Flight Dynamics Analysis System developed for NASA by a commercial contractor.

Compilation Efforts. The Education Program of the SEI began conducting further analysis of the FDAS. The first portion of that effort was to demonstrate that the executable image provided by NASA would execute on the hardware of the SEI. This was successfully demonstrated in July, 1987.

The second effort was to recompile all of the source code provided for the FDAS to verify that it performed in the same manner as the executable image provided. This would afford a check that all of the source code had been delivered with the system. The recompilation of the system and demonstration of the resultant executable was successfully accomplished in August, 1987. It was during this process that certain rather significant problem areas were observed and recorded.

The third effort was to have been the porting of the FDAS from a Digital Equipment Corporation[1] (DEC^{tm}) MicroVAX^{tm} II running under the Virtual Memory System (VMS^{tm}) operating system to a DEC MicroVAX II running under the ULTRIX-32m^{tm} operating system. The SEI had determined that a rather small percentage of their educational affiliates used the VMS operating system in which the FDAS had been developed, while a significant portion used either some form of ULTRIX or UNIX^{tm}[2]. Thus it was determined that the FDAS needed to be ported to the ULTRIX operating system to be useful to the educators affiliated with the SEI.

Some Initial Observations. Upon examination of the more than 25,000 Ada statements (approximately 40,000 lines of code in approximately 277 compilation units), it was found that not a single task was used. Furthermore, only one user-defined exception was declared and it was not used! There were only a few exception handlers and these all used the **when others** clause as the only exception to be handled. There were four generics, two of which had object parameters and two of which had type parameters. The two with type parameters were a generic Stack package and a generic Tree package, both of which were fairly straightforward and conventional. In short, for a piece of code this size, there was essentially no use of the advanced features of Ada!

Identified Porting Problem Areas. The problems that were encountered when the rehosting began were serious and seemed to indicate an overall poor selection of the FDAS for the Ada Artifact. As might be expected with source code developed on the DEC VAX/VMS operating system using the DEC VAX Ada compiler, use was made of the package STARLET. This is an implementation-supplied package which provides access to the system run-time library. Use of this package, which is proprietary, meant that all of these system calls had to be simulated in the ULTRIX environment. It was determined that this was unfortunate, but that it was not impossible to simulate in the ULTRIX environment.

The designers of the FDAS also chose to use INDEXED_IO which is a non-standard input/output package available in DEC VAX Ada. This package would also need to be simulated in the ULTRIX environment. Again, while this was considered an unfortunate choice by the designer, it was not considered impossible to do.

The designers also made use of implementation specific types and objects which are provided in the package SYSTEM by DEC Ada. While these types and objects could have been easily replicated in an intermediate package, their use is what generated the concern. Since the FDAS used expanded names throughout, each occurrence of the package name SYSTEM would need to

[1]DEC, VAX, VMS, and ULTRIX-32m are trademarks of Digital Equipment Corporation.

[2]UNIX is a trademark of AT&T Bell Laboratories.

be examined to determine if the object/type referred to was in the actual package SYSTEM used by other implementations, or if it is one of those which is now found in the intermediate package. Another approach which could be taken would be to place these type/object declarations in the package SYSTEM of the ULTRIX compiler and recompile the entire system including the compiler! While a solution to this problem was clearly possible, it was not very palatable.

The FDAS also used a DEC Ada package called CONDITION_HANDLING which is used in DEC's implementation of the exception handling capability of Ada. Some of the types/objects declared in this package are used in ways which trap the Asynchronous System Traps (AST) of the run-time system. This is an unfortunate deviation by the FDAS system designers from standard Ada exception handling mechanisms. The impact of this use of the CONDITION_HANDLING package was not further studied, but no easy solution seemed evident.

The most disastrous problem with the FDAS was the extensive use of the implementation provided, non-standard, pragma[3] IMPORT_VALUED_PROCEDURE. It was through this mechanism that the designers and implementors were able to access system services directly, *i.e.*, without using the intermediate level facilities provided in the package STARLET. This pragma according to [DEC 85], "allows an Ada program to call an external (non-Ada) routine that returns a result value, but that also causes side effects on its parameters. Because such a routine returns a result, it is analogous to an Ada function; however, a function with **in out** or **out** parameters is not legal in Ada." Thus the pragma IMPORT_VALUED_PROCEDURE maps a function with side effects in the external world (system services) to a procedure in the Ada world. This mapping is necessary because many DEC system services are functions which return their status in their parameters, *i.e.*, they are functions with side effects.

In the case of the FDAS, this pragma allowed the designers to drop down to the operating system to accomplish much of the functionality required in their code which would have been more difficult, perhaps impossible, to accomplish strictly with portable Ada code. Specifically, this procedure was used by the FDAS to drop into the run-time system to spawn additional run-time executives (tasks). This would have been difficult to simulate in the ULTRIX system without some major redesign of the software, since the compiler to be used does not support any similar pragma nor make any other such provisions.

Reactions to Ada. The task of porting the FDAS was not initially considered to be too demanding or too time consuming. In fact, a rather optimistic schedule was proposed to port the system and prepare curriculum materials to facilitate its use as an Ada artifact. Imagine the surprise of many educators when they were told after one month's study that this system could be re-written from scratch faster and cheaper than porting it. Since these educators were all very well versed in software engineering and/or computer science they had no difficulty in understanding the problem, but a great deal of skepticism in relating it to their Ada experience. None of them was intimately familiar with Ada and had mostly heard or read about what this language was capable of doing. They had all heard the hype that Ada was very portable and were therefore very surprised when presented with the facts. Their reactions are not atypical and that is the genesis of this paper.

[3]"A pragma, from the Latin *pragmaticus* (*to order*) is simply a directive to the Ada compiler." [Booch 87A].

Causes of Porting Difficulties

What, then, were the failings of the FDAS which made it effectively nonportable? There were many, so only a few of them will be described. They include lack of proper abstraction, use of implementation dependencies, and use of specific vendor-supplied packages. Ways in which the implementor might have designed the system to avoid these problems will then be discussed.

In fairness to the FDAS system designer, it should be noted that the specifications for the FDAS did not mention portability. It was not a design criteria. Therefore, the designer took advantage of the system services whenever possible, since it is not likely that the FDAS could have provided this functionality in Ada nearly as efficiently as the system services provide it. This also allowed the system to be developed more rapidly and for less cost. Further, it also resulted in less Ada lines of code since system service functions did not need to be replicated at the Ada source code level.

Lack of Proper Abstraction. The first issue is the lack of proper abstraction. "*Abstraction* is one of the basic ways that we deal with complexity; abstraction essentially means to focus on the important elements of some entity." [Booch 87B]. One element of abstraction is modularity. "*Modularity* provides the mechanism for collecting logically related abstractions." [Booch 87B].

In order to successfully port source code for a large system, it is advisable to encapsulate all system dependent features, types, and objects into one specially identified package (or at least some very small number of packages). These packages should contain everything which ties this implementation to a particular compiler or operating system. This makes the job of the persons attempting to port the code less difficult, since they need only re-implement the functionality provided in this/these packages in order to transport the system. Thus, the use of modularity supports portability because when the porting effort is underway, one does not need to search throughout the source code to identify and change implementation specific items. One can refer to the package labeled SYSTEM_DEPENDENT (or some other meaningful name) and have all of the implementation dependent items neatly collected in one place.

The designers and implementors of the FDAS scattered their references to system dependent features, system dependent types and objects, and to vendor-supplied packages throughout their implementation. In order to identify the runtime system calls so as to be able to change them, the persons attempting to port this system would be required to search throughout the entire source code for identifiers which have an embedded '$', which DEC uses to mark their system calls. This is an unnecessary waste of time and effort directly attributable to lack of proper abstraction during the design process.

Use of Implementation Dependencies. The next issue that must be considered is the issue of using implementation dependencies. These often take the form of special "hooks" to the underlying operating system services. This is almost sure to be proprietary and will be exceedingly difficult to reproduce on another system, even if it is encapsulated in a special system dependent package. The use of these hooks to provide functionality to the software is understandable since it allows the code to be released more quickly and is probably more efficient, given its implementation directly in the underlying operating system. However, the use of such mechanisms should be used advisedly, if at all, because the reproduction of that functionality in a system to which the software may be ported will be very difficult.

The FDAS made extensive use of these dependencies as discussed above. Specifically, the FDAS made use of the pragma IMPORT_VALUED_PROCEDURE to access system services. It also used objects and types defined in the package SYSTEM, which are specific to the DEC VAX Ada implementation. Finally, the FDAS made use of the special types provided in the package CONDITION_HANDLING to avoid the use of Ada exceptions.

Use of Vendor-Supplied Packages. The last issue is the use of specific vendor-supplied packages. While this is not necessarily a bad design decision, it is nonetheless unfortunate when portability or reusability is desirable. It means that the functionality of the vendor-supplied packages must be re-written and re-tested. It also means that if the vendor-supplied software used some low-level access to special routines which are proprietary, then there may be considerable difficulty in providing the same functionality.

In the case of the FDAS, the designers and implementors made extensive use of special vendor-supplied Input/Output (I/O) packages. Ada specifies three different kinds of I/O, *viz.*, text, direct and sequential. Vendors, using the provisions of Ada for extensibility, are free to provide additional I/O varieties. However, the use of these special I/O packages should be carefully studied. Consider unsuspecting Ada novices that blithely use a vendor-specific I/O package in their development and then attempt to move this implementation to another machine. Imagine their surprise in finding that all I/O must be redone! While there is certainly nothing illegal about supplying or using vendor-specific I/O packages, they should be used advisedly.

Ada Support for Porting Software Systems

Design Support. Ada provides the language support necessary to minimize the problems associated with the portability of software systems. Specifically, it provides an encapsulation mechanism, called a **package**, which can be used to concentrate all system dependencies into one location (be honest, you must acknowledge that any large system is likely to have system dependencies). This means that programmers attempting to port a software system will only need to concern themselves with this package(s), and not have to scan all of the source code searching for system dependencies. This makes the job of the people porting the system much less complex.

It must be clear from this that it is the design of the system which is of paramount importance since that is where the encapsulation occurs. Notice that the language provides the support mechanism needed, but does NOT actually make the system portable. Thus, we cannot say that Ada is portable, but that it SUPPORTS portability!

This is not an indictment of the programming language Ada. It is important to emphasize that Ada provides excellent facilities and features for portability, but these features will not "automagically" guarantee portability. The human element in the system design is the most important element in portability. Unfortunately, there is nothing Ada can do to directly provide this, but it can assist. Ada is readable, has a regular structure, and is rich in expressive power due to the many various kinds of typing mechanisms provided.

Rule of Thumb #1: Humans must design systems for portability; it does not happen "automagically".

Design Decisions. Portability is the responsibility of the system designer, not the programming language in which it is implemented. Designers of software systems must often choose between many options and weigh the various benefits versus costs of many design decisions. It is imperative that portability be considered in these decisions in as much as it is often the case that systems which are in use are often ported, even though there was no original inclination to port the system and thus it was not part of the design criteria. This is a simple fact of modern software life! Consequently, decisions which will adversely affect portability must be deliberate and must be carefully weighed.

Rule of Thumb #2: Design all of your systems as if you personally will have to eventually

port it to the most diametrically opposed machine and operating system.

Standardization. Ada has made great strides in standardization. For once in history, we had the complete design of the language BEFORE we had any implementations. This allowed the designers to be rather academic about their design. However, the designers were also pragmatic. They realized that it is not possible to completely specify every aspect of the language due to variations in hardware, variations in implementations strategies, and variations in the machine instructions available on different hardware. They therefore allowed implementation dependent aspects to the language.

In particular, since the language was specifically designed for extensibility, they allowed for variations between implementations by the mere fact that some vendors may chose to provide additional packages which may prove useful to their customers, which are not specified by the language. Presumably, the same functionality could have been written by the user, and the implementor is merely saving the user some effort and boosting chances for a sale.

But it is not always that simple. Some vendors provide packages which allow users to have some functionality which they would not be able to write themselves. This is a particularly dangerous facility to use if portability is to ever be attempted. Specifically, the DEC VAX Ada implementation provides a pragma called IMPORT_VALUED_PROCEDURE which was described earlier in this paper. The vendor satisfied a perceived need of the user by providing a pragma which could be used to access DEC system services, but this is clearly not a transportable concept.

"Standardization [by International Standards Organization (ISO) and/or American National Standards Institute (ANSI)] continues to be a major impetus for improvement of programming language portability. Unfortunately, most compiler designers succumb to a compelling urge to provide 'better' nonstandard features for a standardized language. If portability is a critical requirement, source code must be restricted to the ISO or ANSI standard, even if other features exist." [Pressman 82].

Rule of Thumb #3: Use only the standard features of the language; do not be seduced into using vendor-specific features.

Attributes. Another mechanism in Ada which can be used to assist in porting software systems is the use of attributes. Attributes are essentially predefined functions which return values representing either predefined quantities or certain static features of types, or which interrogate the system about implementation dependent values. For example, there is an attribute called P'FIRST which "for a prefix P that denotes a scalar type, or a subtype of a scalar type: Yields the lower bound of P." [LRM 83]. Thus, if some software system makes use of a procedure to which many different subtypes are passed at different times during execution, then within the procedure the programmer can access the lower bound of each subtype dynamically, *i.e.*, without making use of some constant, probably artificial, value. Ada has many attributes available to the programmer.

Rule of Thumb #4: It is a good practice to use attributes whenever possible rather than specific values.

Underlying Runtime System. Each Ada compiler implementor provides their own implementation of an Ada runtime system. Each of these systems is unique and there is no guarantee that programs successfully operating under one runtime system will function in the same manner under another runtime system. In particular, some of the differences are in task scheduling schemes, *i.e.*, time-slicing versus run-until-blocked. Another is the implementation of varying priorities for tasks which should only be used to advise the compiler about the relative importance of tasks, not to provide scheduling.

Rule of Thumb #5: Always design your system so as to provide your task scheduling in code; do not rely upon pragmas which set runtime system priorities.

Ada Compiler Validation Capability. The Ada Compiler Validation Capability is a series of test programs which every Ada compiler must pass in order to use the certification mark as a validated Ada compiler. These tests are designed to ensure compliance by the vendor to the Ada Language Reference Manual, which in some sense, represents the specifications of the Ada programming language. Each year more tests are added to attempt to minimize the divergence from the standard of compilers which are released and validated.

However, it clearly is impossible to verify that everything is implemented absolutely correctly in every instance. Therefore, the ACVC cannot imply anything about those parts of the language which are not tested. In fact, about all that the ACVC can do is to increase the user's level of confidence that the Ada compiler that is being used is probably correct. While that may not be much, it is certainly much more than is guaranteed with most other languages.

It is important to emphasize that the ACVC does not guarantee portability. This is true for at least two main reasons. Firstly, the test suite does not, and probably will never, test every nuance of the language which might be important for portability. Therefore, some aspects of one implementation may be different from another implementation, adversely affecting portability.

Secondly, there are several items in the Ada language which were deliberately left unspecified, *i.e.*, implementation dependent. In every case, different implementations have chosen to implement these items in ways which best suited their needs. This implies that many of the implementatic do the same thing in quite different ways. In general, this is not bad. However, there are some portability issues which are affected by the differences in implementations.

Note that when an implementation decides on its implementation dependent values, *etc.*, these must be documented in Appendix F to their version of the Ada Language Reference Manual. It is there that the users can find the answers to their questions about the different implementation dependent values, *etc.* While this aids in the identification and isolation of items which must be segregated into implementation dependent packages to facilitate portability, it also demonstrates what is meant by the lack of complete portability in Ada.

Rule of Thumb #6: The ACVC is a good indicator that your compiler meets the standard; it is not a guarantee.

Use of Style Guides. All of the foregoing discussion naturally leads to the question, are there things which system implementors can do to enhance the portability of their systems? The answer is absolutely! There have been several studies which support this statement, notably [CSC 87], and [Seidewitz 87]. These documents provide extensive discussions about good style techniques which will enhance portability.

One very good example of a formalization of the ideas that are discussed in this paper, plus many more, is [Seidewitz 87]. This document represents a codification of the style issues which contribute to effective software engineering practices and thus supports portability of the resultant code. It must be recognized that not all code produced in accordance with this style guide will necessarily be portable, but the guide at least provides a basis for assisting in the portability of software.

Rule of Thumb #7: Decide on well-considered Ada Style Guidelines and ensure that programmers comply with them.

Conclusions

It was the purpose of this paper to emphasize that Ada has no magic inherent properties which make it any more portable than any other programming language. It does have several integrated, well conceived features which facilitate portability. However, ultimately, the degree of portability of a system is much more a function of its design than the language in which it is implemented. Thus, the statement that Ada is portable is not only a myth, it is a lie!

References

[Booch 87A]	Booch, G., *Software Engineering with Ada*, Second Edition, Benjamin/Cummings, Menlo Park, 1987.
[Booch 87B]	Booch, G., *Software Components with Ada*, Benjamin/Cummings, Menlo Park, 1987.
[CSC 87]	"Ada Reusability Handbook", Computer Sciences Corporation, Technical Report SP-IRD 11, *proprietary*, 1987.
[DEC 85]	*VAX Ada Language Reference Manual*, Digital Equipment Corporation, AA-EG29A-TE, 1985.
[Lecarme 86]	Lecarme, O.,Gart, M.P., *Software Portability*, McGraw-Hill, New York, 1986.
[LRM 83]	*Reference Manual for the Ada Programming Language*, U.S. Government, 1983.
[Pressman 82]	Pressman, R.S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, 1982.
[Seidewitz 87]	Seidewitz, E., *et. al.*, "Ada Style Guide (Version 1.1)", SEL-87-002, Goddard Space Flight Center, NASA, Greenbelt, Md., 1987.
[Wiener 84]	Wiener, R., Sincovec, R., *Software Engineering with Modula-2 and Ada*, John Wiley & Sons, New York, 1984.

This Page Left Blank Intentionally

Ada Education for Managers: Defining Needs and Meeting Requirements

Captain Jeanne L. Murtagh
Wright Patterson Air Force Base

This paper was not received in time to be included in the proceedings.

This Page Left Blank Intentionally

Teaching Ada in the University

Kathleen Warner
Department of Computer and Information Sciences
Marshall University, Huntington, W.V.

Abstract:

Few changes in University level curriculum have stimulated as much support and at the same time, created as much controversy as the teaching of the Ada programming language. College Administrators, Politicians, Business and Community Leaders and Computer Science Educators all agree on the importance of including Ada within University curriculum offerings. The controversy concerning the introduction of Ada to University level programming language offerings is not with whether Ada should be taught within the University, but with how it should be taught.

Background:

The support of the Department of Defense and the wide reaching effect of that support is recognized - software engineering and software development will be influenced by the existence of Ada for at least the productive lifetime of currently enrolled students. There is a keen awareness that selected sectors of computer and software development industries will be directly effected by the existence of Ada in much the same way that sponsorship of COBOL influenced programming practices and course offerings over the past twenty-five years.

Programming practices and Computer Science programs have been influenced and changed in other ways as well. Dijkstra's 1968 letter to the editor of the Communications of the ACM, "Go To Statement Considered Harmful" made the Computer Science community aware of the need for structured, reliable, programs. Since the early 197Ø's, computer scientists and programmers have followed a body of programming methods and techniques called "structured programming."

The corresponding recognition of the "Software Crisis" contributed directly to the development and evolution of software engineering as a discipline, separate and distinct from computer programming. Programmers began to shift the focus of their attention from basic language concepts, systems and their implementation to the construction of systems from discrete program modules (Haberman, 1986, p.29).

The Ada programming language evolved in conjuction with the discipline of software engineering. It is a language which directly incorporates and supports the modern software engineering principles of abstraction, information hiding, modularity, locality, and reusability. The use of Ada can lead to the realization of the software development goals of readability, clarity, reliability, efficiency, clarity, maintenance and portability of software products.

Ada is unique. In constrast with programming languages which were developed either by individuals or by committee, Ada was created by a language design team. Public commentary was solicited by Ada's designers and received from academic and industrial experts, Department of Defense personnel and other intended users of the language prior to its standardization. Sammett (1986) remarked that Ada stands apart from other programming languages in its support of software engineering principles, its standardization and compiler validation, and its ability to create portable code.

Ada in the University Curriculum:

Computer Science educators generally recognize and accept that successful teaching of beginning programmers requires the use of small, well structured, readable languages (Burd, 1986). Pascal, which fits all of these requirements, is frequently the language of choice in Computer Science programs following an ACM recommended curriculum. Fortran, subsets of PL/I, (to keep small the language and the syntax rules to be learned) even BASIC and COBOL are used to instruct students in programming practices and techniques.

Ada is slowly being integrated into Computer Science curriculum offerings. Burd (1986) noted that Ada is often introduced in specialized courses such as Software Design or Software Engineering classes. Sammett (1986) observed that "one of the key aspects of Ada is its usefullness for new types of software education." She remarked that Ada has been successfully used in teaching specialized courses in numerics, concurrent processing, data structures and can be successfully used as a first class in programming (1986).

As Computer educators develop course offering in Ada, they must address the following issues:

1. Where does Ada instruction fit within a Computer Science curriculum ?

2. Can Ada be adequately taught as a first programming language or should the teaching of Ada follow a foundation preparation in a block structured language such as Pascal or Modula-2 ?

3. With the external pressures to add Ada to university curricular offerings, how will Computer Science instructors be prepared to address this curriculum expansion and teach Ada ?

There are other concerns as well. For example, if Ada is taught as a first, and stand alone language, how many courses must be included in the curriculum to adequately cover the extensive and powerful features of the language ? Can students be exposed to the depth and breadth of the Ada language through small to medium scale programming projects ?

Ada Education in West Virginia:

Throughout West Virginia, there has been a drive to add Ada to College and University curriculum offerings, both because Senator Robert Byrd has invested so much time, effort, and personal attention to the sponsorship of Ada and the Software Valley Corporation and because educators recognize the need to prepare our students well for entry into the software development labor force.

At the present time, Ada is taught at seven institititions of higher education in West Virginia. These institutions are regionally well distributed, represent colleges and universities from both the public and private sector, and computer science programs offering both two year and four year degrees.

West Virginia University

Ada education began at West Virginia University located in Morgantown, W.V. under the direction and leadership of Dr. Joan Van Scoy. Dr. Van Scoy worked with one of the first validated Ada compilers and began to develop an Ada course in 1984. She has been active in teaching Ada to graduate students enrolled in the Master's Degree program at WVU and has helped disseminate knowledge about Ada to other Computer Science educators and professionals through partìcition in panels, seminars and workshops held throughout the state.

Beginning in the fall, 1988, West Virginia University will offer three classes in Ada. After successfully teaching Ada as a first programming class in an experimental setting in the summer of 1987, the faculty elected to offer a choice between Ada and PL/I as the first programming class to students enrolled in the undergraduate degree program in Computer and Information Sciences. An upper division Ada class in Software Engineering Techniques and Concurrent Processing will be available to upper level majors and graduate students. Effectively, WVU will offer the ACM equivalent of CS/1 and CS/2 through the use of Ada.
Ada programs have been written using both VAX/Ada supported by WVNET and through use of the Alsys compiler written for use on IBM/PC's.

Alderson-Broaddus College

Alderson-Broaddus College, a private, four year liberal arts college located in Philippi, W.V. currently offers two Ada classes. Both of the Ada classes are open only to upper division students majoring in Computer Science. Students may enroll in an Ada programming class after successfully completing two semesters of Pascal and a Data Structures class, and Software Engineering Class with Ada after completion of the introductory Ada class.

Alicia Kime, who teaches the Ada language class, studied with Dr. Van Scoy as a graduate student at WVU. Gary Schubert, who teaches the Software Engineering and Ada class, together with Alicia and other members of the Alderson-Broaddus CS faculty also studied Ada during an intensive two week seminar/workshop conducted for them by Ada-Soft Corporation. They reinforced their knowledge of Ada through intensive self-directed study following the seminar.

They concur that the proper placement of Ada in Computer Science offerings is after the student has been thoroughly grounded in a block structured programming language such as Pascal or Modula-2 and successfully completed a Data Structures class.

West Virginia Northern Community College

An Ada programming class implemented as an ACM CS/1 course is currently taught by Robert Terry, an instructor at W.V. Northern Community College. WVNCC is a public, two year college located in the Wheeling-Wierton area, in the northern part of the state. Students are admitted to the Ada class after successfully completing a programming class in another language and receiving permission of the instructor, thereby limiting enrollment. Approximately half of the students who have completed the Ada class are experienced programmers who work in local industries.

Before introducing Ada as a formal class, it was offered as a special topics class. Course material was refined and introduced as part of the regular course offerings in the spring, 1988. A second Ada class is being planned and is under development. This class will be implemented as an ACM CS/2 programming class. WVNCC has used Putnam Texel's, "Ada: Programming with Packages" as their class text.

Mr. Terry completed a Master of Science Degree in Computer and Information Science at WVU and is also a former student of Dr. Van Scoy. The introductory Ada class emphasizes the use of Packages and the strong typing features available through the language. Students are introduced to the concept of Generic packages and learn about instantiation through the use of standard I/O procedures. Class assignments are programmed using the Janus compiler.

Because of their location in an industrialized region of West Virginia, WVNCC is in a unique position to offer Ada training to employees of local industries. Wierton Steel Corporation recently purchased a blast furnace from General Electric Corporation. The processing control programs for the blast furnace are programmed in Ada.

W.V.College of Graduate Studies

The West Virginia College of Graduate Studies (COGS) is located in the Kanawha Valley approximately 15 miles from Charleston, West Virginia, the state capital. Only Masters level course work is offerred at COGS. Two Ada classes are currently taught. The first class is an introduction to Ada, the syntax, semantics and specialized features of the language. The second Ada class is essentially a software engineering class which makes use of the features of Ada which support software engineering principles and practices.

Robert Hutton is the instructor for these classes. Bob is an experienced programmer and systems analyst who worked in the Kanawha Valley chemical industry before becoming a full time member of the COGS staff. Programming is performed using the VAX/Ada compiler supported by WVNET.

Since COGS offers only graduate level course work, students who enroll in the Ada classes will have completed extensive course work in computer language classes, data structures classes and other classes which are considered a part of standard computer science curriculums. Many of the students enrolled in the COGS program are actively employed as computer professionals.

Bob has expressed the opinion that it is difficult to teach students "programming-in-the-large" with small scale projects. His opinion on the placement of Ada within Computer Science program offerings is reminiscent of Sammett, "... Ada has been and should be used as a vehicle for teaching software engineering principles in both academic and industrial settings (1986, p.129)." Textbook choices have paralled this philosophy, both Cohen, "Ada as a Second Language" and Booch, "Software Engineering with Ada" have been used as course texts.

West Virginia Technical College

W.V. Tech has recently added an Ada programming class to its course offerings.

Marshall University

Ada was first introduced to the Computer Science offerings at Marshall University as a Special Topics class in the spring, 1987. I conducted the class on an experimental basis and introduced concepts and principles of software engineering as well as syntax and semantics of the Ada and Modula-2 programming languages.

External procedures, packages/modules, and data typing were a focus of study. Students were introduced to the concept of generic subprograms but did not write any generic code. They learned about instantiation through use of standard I/O procedures in both programming languages. The class used Sincovec and Weiner, "Software Engineering with Ada and Modula-2" and Booch, "Software Engineering with Ada" as required texts. DEC Vax/Ada and Logitech Modula-2 reference manuals were used as lab supplements.

The students enrolled in the first special topics class had completed two semesters of PL/I or Pascal, Data Structures, and Programming Language Structures. These students had first been introduced to Modula-2 and Ada in the Programming Language Structures class which I had taught during the previous semester. Many of the students enrolled in the class had attended Ada Expo 1986 in Charleston, W.V. and were enthusiastic about the new class.

Student demand for the class in the spring semester exceeded available space in the class and it was re-offered as a Special Topics class in the fall, 1987.

It was decided to drop Modula-2 from the class material because of the difficulties experienced in running the Logitech compiler in a PC lab equipped only with floppy disk drives. Ada language features received greater emphasis in this class. More attention was given to separate compilation units, exception handling, program libraries, and the concept of reusable code.

Students in this class also had completed two semesters of either PL/I or Pascal, Data Structures, and the Programming Language Structures class, where they had been minimally exposed to the Ada programming language. Over half of the students enrolled in the second special topics class had completed the first class in the previous spring semester.

For better class management, students were divided into programming teams. Students who had completed the first special topics class chose term projects and worked on these projects throughout the semester. We held weekly conferences to insure that they continued to make progress on their projects. One team implemented a Spell-Checker package in Ada; a second team produced a compiler written in Ada to produce 68000 machine code.

A third group was composed of students with minimal previous experience or knowledge of Ada. I taught this group of students, covering the syntax of Ada on an accelerated basis. This group of students also completed term projects which were assigned by the instructor.

After demonstrating an understanding of Ada syntax, and the concepts of packages, procedures, and separate compilation units, they completed two group projects. The projects were a simple Math package and a Statistical package. I provided the package and procedure specifications, and individual students completed the required procedures.

These students were also required to write generic procedures for the creation of standard data structures such as stacks, lists, queues, and trees along with the algorithms for manipulating, searching, and updating these structures. This was a useful and successful exercise because it worked from a topic known to the students - data structures and their manipulation, while creating an appreciation and understanding of the usefullness of generic procedures and packages.

Programming assignments were completed using the VAX/Ada compiler supported by WVNET. The required text for this class was Watt, Wichmann, and Findlay, "Ada Language and Methodology." Supplementary materials included Booch, "Software Engineering with Ada" and Vax/Ada language reference manuals.

Software Engineering principles were not de-emphasized, but applied within the context of the Ada programming language. The class instructor received training in Ada at seminars conducted by Software Valley Corporation in Morgantown, WV., and through the Software Engineering track at Ada Expo, 1986, in Charleston, W.V., as well as intensive, self-directed, hands-on practice.

The Department of Computer and Information Science plans to implement a Software Engineering and Ada class open as an elective class for upper division computer science majors and graduate students from the Mathematics and Physics departments. Formal course proposals have been submitted to the college and university curriculum committee for review and approval.

Beckley College

Beckley College is a private, two year institution located in Beckley, West Virginia in the Southeastern region of the state. Beckley College supports a 2+2 transfer program with West Virginia Tech. Many of their students will go on to a four year degree program after completing a two year degree.

Currently, one Ada programming class is offered and students may enroll only after they have completed course work in another programming language. At the present time, students may enroll in the Ada class after completing a class in Fortran, BASIC, COBOL, or Pascal. In the fall of 1988, students will be required to complete a semester of Pascal before enrolling in the Ada class.

Stephanie Ketz, the class instructor received training in Ada by attending seminars conducted by Penn State University, ASEET, and through focused, self-directed, hands-on practice. She has emphasized the use of separate compilation units, packages, data typing, and exception handling in her coverage of the language.

Programming assignments are completed using the Meridian compiler supplied through Data General Corporation. Beckley College has used Putnam Texel's "Ada: Programming with Packages" as the required class text.

Summary and Conclusions:

While all faculty currently teaching Ada in West Virginia agree on the importance of including it in our computer science offerings, there is a diversity of opinions regarding the proper placement of Ada within a computer science curriculum.

In West Virginia colleges and universities, there is much variety in how the Ada programming language is taught. At some institutions it is taught as a first programming language; others introduce Ada as a second programming language, or as a language which supports and complements teaching software engineering principles or specialized classes such as Concurrent Processing. At COGS, Ada is taught only at the graduate level.

Institutions teaching Ada as a first programming class, or as a class open to students who have completed a Fortran, BASIC, or COBOL class find Putnam Texel's "Ada: Programming with Packages" a useful text. This text is readable and understandable to their students. Introductory Ada classes typically omit advanced features of the language (e.g., exception handling, generics, tasking) and focus on more easily learned concepts and needed features. Burd (1986) asserted that "one feature which cannot be ignored in an introductory course is the Ada package" and all of the institutions discussed in this paper devote considerable time and attention to Ada packages, procedures, and parameter passing details.

Ichbiah (1984), Sammet (1986), Burd (1986), Booch (1988) and other computer professionals have stated that Ada is an easy to learn language for those with previous experience with block structured languages such as Pascal or Modula-2. The syntax of Ada is similar enough to Pascal that Burd (1986, p. 515) observed "the experienced Pascal programmer who knows nothing about Ada will not have trouble reading a simple Ada program." Ichbiah (1984, p.997) stated "People who have already learned a structured programming language like Pascal are probably going to learn Ada in a very short time. For a Pascal programmer to reach an equivalent level of programming in Ada will take less that a week."

The faculty teaching Ada as an advanced programming class or as part of a software engineering class where enrollment is limited to students with previous experience in block structured languages concur with these observations. I think none of us will claim that our experienced Pascal students reach an equivalent level of mastery in Ada within one week's time. However, we have found that coverage of more advanced features of the language is facilitated by having the students complete two semesters of Pascal and an additional data structures class before attempting to learn Ada. The choice of texts (Booch, Software Engineering with Ada; Cohen, Ada as a Second Language; Watt, Wichmann, and Findlay, Ada Language and Methodology; Gehani, Ada: An Advanced Introduction) parallels the expectation that these students will be able to participate in an accelerated or advanced learning process.

Almost all of the institutions currently teaching the Ada programming language first introduced it to their curriculum as a seminar, special topics class, or experimental class. Development of formal classes added to standard course offerings evolved from the refinement of these special topics classes.

Many of the faculty teaching Ada learned about how to foster the use of software engineering concepts and the Ada language by attending seminars conducted by software corporations specializing in Ada education. Knowledge and understanding of the language was reinforced and extended through intensive and self-directed practice and study. Several of the faculty currently teaching Ada completed formal classes in the language at WVU or at COGS.

All faculy teaching Ada in West Virginia are members of the Software Valley Corporation. Although there are differences in how we have chosen to introduce Ada to our course offerings, we share an excitement and enthusiasm about teaching Ada. Collectively, we are working to improve our Ada classes through discussion, meetings with members of Software Valley Corporation, and colloquiums conducted with other West Virginia Computer Science educators. We have discussed the possiblity of establishing a "West Virginia Ada Educators" group to coordinate Ada language instruction throughout our state.

References

Booch, G., Software Engineering with Ada, Benjamin Cummings Publishing Company, Inc., Menlo Park, California, 1987.

Burd, B., Teaching Ada to Beginning Programmers, Proceedings of the ACM, 1986.

Cohen, N., Ada as a Second Language, McGraw-Hill Book Company, New York, New York, 1986.

Gehani, N., Ada: An Advanced Introduction, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1984.

Haberman, A. N., Technological Advances in Software Engineering, Proceedings of the ACM, 1986.

Ichbiah, J., " Ada: Past, Present, Future: An Interview with Jean Ichbiah, Principal Designer of Ada", Communications of the ACM, 27,10, pps. 990-997, Oct. 1984.

Sammet, J., "Why Ada is Not Just Another Programming Language", Communications of the ACM, 29,8, pps.772-733, Aug. 1986.

Watt, D., B. Wichmann, and W. Findlay, Ada Language and Methodology, Prentice-Hall International, Englewood Cliffs, N.J., 1987.

Weiner, R., and R. Sincovec, Software Engineering with Modula-2 and Ada, John Wiley and Sons, Inc, New York, 1984.

EXPERIENCES WITH ADA SUBSETS IN AN UNDERGRADUATE COMPILER COURSE

By *J. D. Wethington*
Associate Professor Computer Science

Mesa College, Grand Junction, CO 81501

ABSTRACT

This paper details the use of a subset of Ada* in an undergraduate compiler course. The author has taught this course several times. However, the Fall semester 1987 was the first time he used a subset of Ada as the source language for translation. The course is a three semester hours undergraduate course open to juniors and seniors at Mesa College. It is a required course for Computer Science majors. In order to make the size of the source language suitable for a one semester project, the author used a subset of Ada. Throughout the course , the author and the students referred to this subset as 'Mesa_Ada.' Although the author agrees with the concept of not allowing subsets of Ada to be used in production programming, he has found it impossible to work within this constraint in a student environment.

The target language for the translation was code for a stack machine. As Barrett (1986) was the text book used in the course, we used his stack machine. Taking examples from 'Mesa_Ada,' we covered the standard material found in beginning compiler courses. The author chose the subset of Ada for two reasons. These were (1) it is an adequate source language for the course and (2) he wanted to introduce his students to Ada.

The results were most encouraging. Most of the students completed the semester project. All of them learned the introductory concepts of compiling, and all of them learned the basic structure and syntax of Ada.

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

A. Introduction

Mesa College is a Baccalaureate Institution located in Western Colorado. It offers a major or emphasis in Computing Science that closely follows the ACM recommended curriculum. One of the required courses is CSCI 450 Compiler Structures. The course is open to juniors and seniors. It covers scanning, parsing, translation and symbol table management. Also, an integral part of the course is a semester project. The instructor divided the students into groups of two or three people. Each group wrote the front end of a compiler. The source language was Mesa_Ada, a subset of Ada, and the target language was the P-code for a stack machine.

This paper discusses Mesa_Ada, the stack machine, sample translations and the results obtained in the course. The main point we would like to make in the paper is that only subsets of Ada are usable in an educational environment. The complete language is just too large to use in one semester courses.

After some thought, we divided the project into four phases: (1) the scanner, (2) the symbol table routines, (3) the parser and (4) semantic routines to generate P-code. We used finite state machines to design the scanner that recognized the tokens of Mesa-Ada. As the students had previous experience in this area, this proved to be the easiest part of the project. We organized the symbol table as a stack of binary search trees. It also did not present the students with any difficulty. Since the grammar was close to an LL(1) grammar, we developed the parser as a recursive descent parser. After suitable discussion in class, the students were able to write it. However, the last phase, the code generation phase, presented a challenge to the students.

B. Ada Subset Used

As stated above, the language had to be abridged before we could use it for a semester project. In past semesters the author used subsets of other Algolic languages. Hence, he chose a subset of Ada that was close to what he had successfully used before. Figure 1 shows the top view of a program.

program

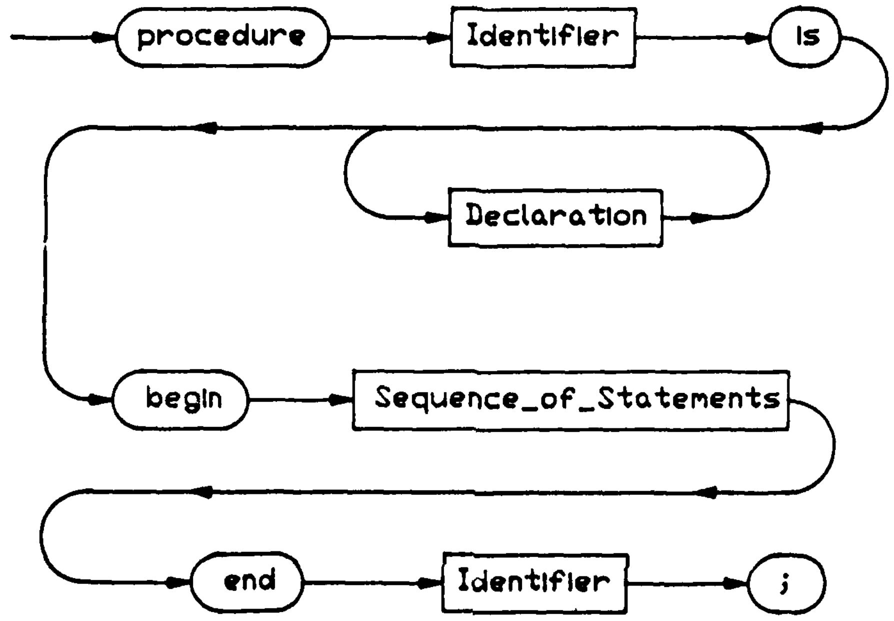


Figure 1. Syntax Diagram of a Program

We will not list all of the syntax diagrams that we used to define the source language for the students. However, several of them will be needed to describe what we did in the course. We will list some of these. Also, we restricted the numeric types to integer numbers, and we excluded strings of characters. Figures 2, 3 and 4 show some of the syntax diagrams for the object declarations that we included in Mesa_Ada.

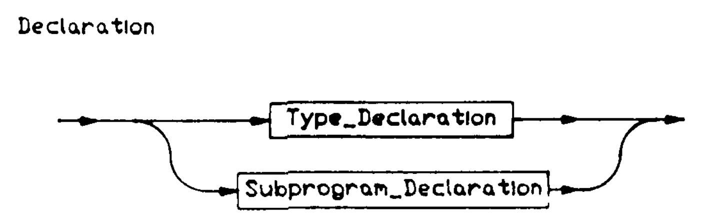


Figure 2. Syntax Diagram for Declaration

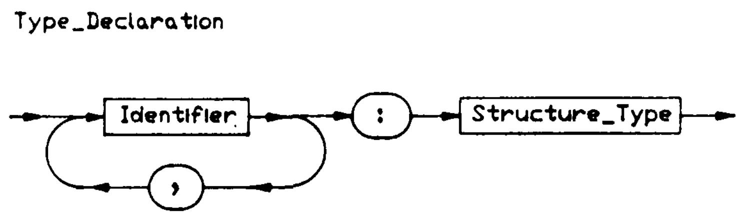


Figure 3. Syntax Diagram for Type_Declaration

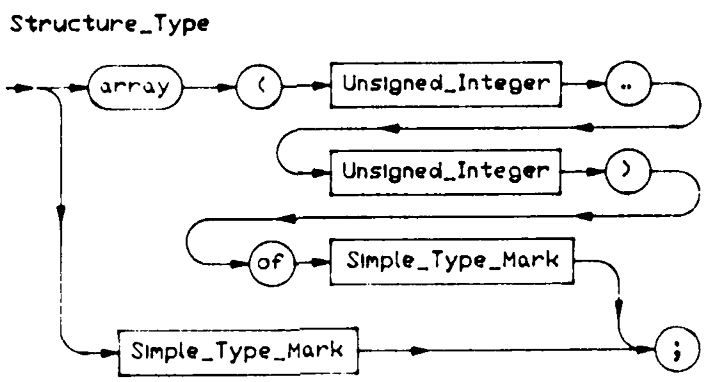


Figure 4. Syntax Diagram of Structure_Type

We used these syntax charts to parse the declarations, and then make the appropriate entries in the symbol table. This was the point in the course where the students began to appreciate their problem.

As we spent a considerable effort on subprograms, we did need the syntax of a subset of them. The design of the symbol table, and the translation routines themselves followed from what we chose to include. Figures 5 and 6 show part of the diagrams for functions and procedures that we used in the course.

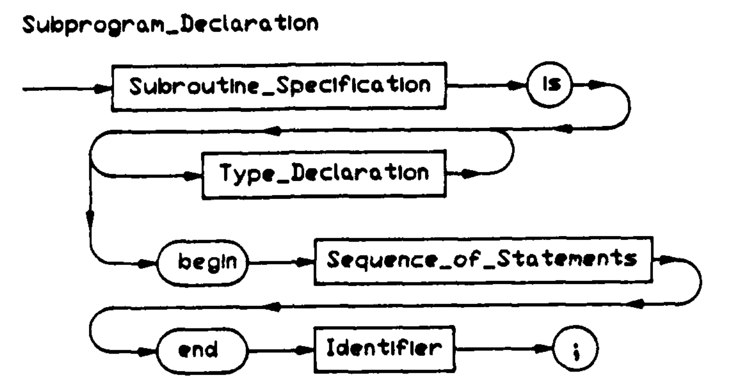


Figure 5. Syntax Diagram for Subprogram_Declaration

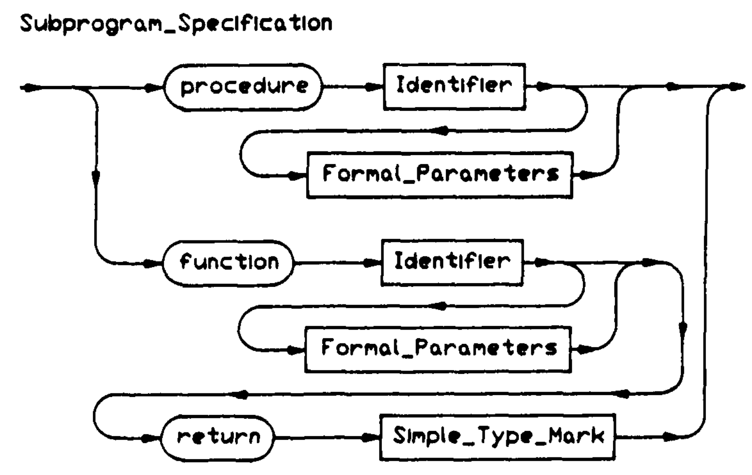


Figure 6. Syntax Diagram for Subprogram_Specification

When we considered statements and expression, we realized there would need to be a heavy abridgment of Ada. As we did the abridgment, past experience dictated what was needed in the course. Although it does not show all of the statements with which we worked, Figure 7 does show the statements that we wish to discuss in the paper.

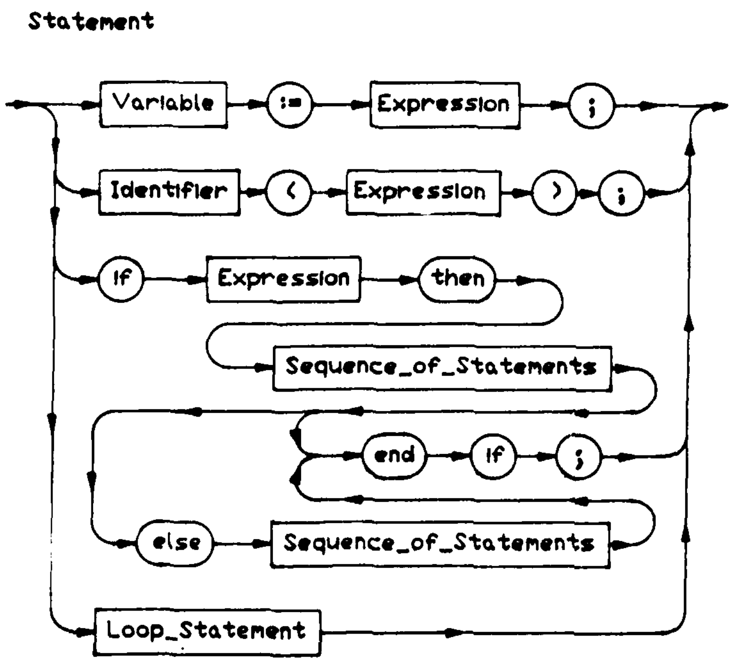


Figure 7. Some Statements discussed in this Paper

Expressions were another area that needed heavy abridgment before it was feasible to use them. Basically, we excluded all of them except some of the relational, arithmetic and boolean expressions. We do not see the need to show the syntax, but we will discuss it as needed to develop the rest of the material on statements.

C. Stack Machine Used

As we stated above, the stack machine that we used in the course is the one found in Barrett (1986). It consists of the following parts (1) a program store, (2) a run time stack and (3) an interpreter for the P-code instructions. Figure 8 shows an example program. Figure 9 shows a translation of this program into P-code instructions. Figure 10 then shows a schematic diagram of the machine. The machine is simple, but it is sufficient to implement every thing that is needed in a first course in compiling.

```
          procedure Example_1  is          -- the program
          A , B : Integer  ;               -- does nothing

          begin                            -- but provide
             Get ( A ) ; Get ( B ) ;       -- an example
             loop                      -- for translation
                exit when A >= B ;
                A := A * B + 5 ;
                Put ( A ) ;
                Get ( A ) ; Get ( B ) ;
             end loop ;
         end Example_1 ;
```

Figure 8 Example Program.

This code can be translated into the following version of P_code. Again, it is patterned after the one in Barrett (1986).

```
          0  INCS    2          -- save space for A and B
          1  READI              -- Read A
          2  STOR I  0 1            -- and
          3  READI                  -- then
          4  STOR I  0 2            -- Read B
          5  LOC  L1
          6  LOAD I  0 1            -- is
          7  LOAD I  0 2            -- A greater than
          8  LES  I                 -- or equal B?
          9  FJP  L2                -- no then jump out
         10  LOAD I  0 1            -- yes
         11  LOAD I  0 2            -- then
```

```
12  MPY  I                    -- calculate
13  LOAD =5                   -- A*B+5
14  ADD  I                    -- and
15  STOR I   0 1              -- store it in A
16  LOAD I   0 1              -- and
17  CXP      PRINTI           -- then
18  INCS     -1               -- write it out
19  CXP      READI            -- Read A
20  STOR I   0 1              -- and
21  CXP    READI              -- then
22  STOR I   0 2              -- Read B
23  UJP      L1               -- jump back
24  LOC    L2
25  EXIT   0
```

Figure 9. Translation of Example_1.

Below we show the diagram of the stack machine. The C array, the stack, is shown as it would look just after the execution of the instruction 'INCS 2.'

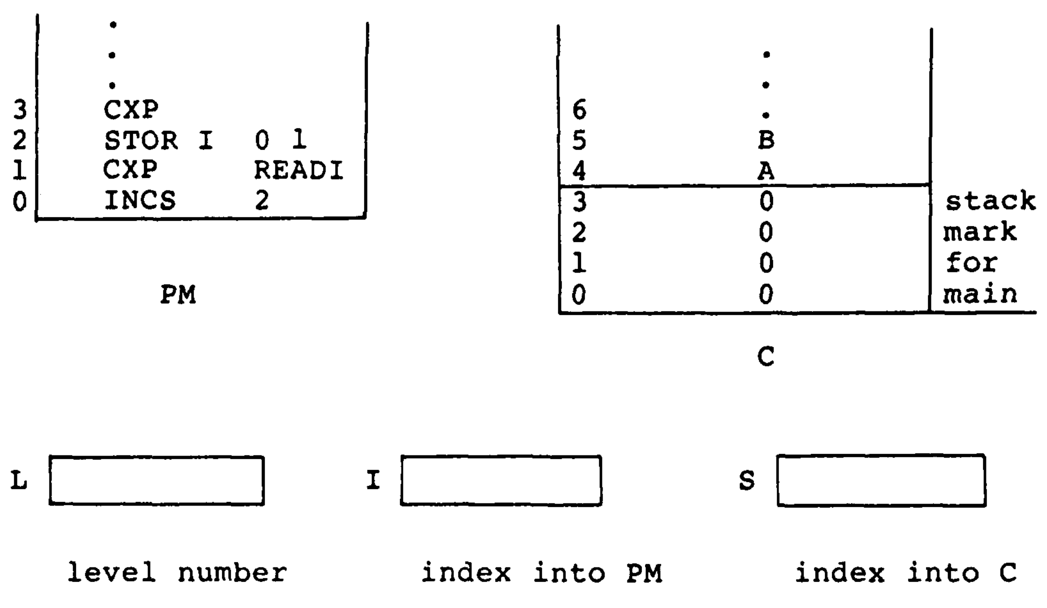


Figure 10. Schematic diagram of the P-code machine.

A simulator must first load the P-code into the PM array, and then initialize the stack C. It then enters a fetch cycle. Then, it fetches and executes each instruction in turn. A working simulator was not part of the course. This course concentrated on the problems of translation. The students wrote the simulator for the machine in a pre-requisite course on programming languages.

D. Sample Translations

Below, in Figure 11, we show the program that the students' compilers translated for the final submission of the project. As we stated above, most of the students were able to write a recursive descent compiler that translated this program into P-code. As the program translated into about 100 P-code instructions, we will not show the complete translation. However, Figures 12 and 13 show representative translations.

```
Procedure Clean_Final is

A , C  , B : Integer ;
E , F1 , G : Boolean ;

procedure P1 is
A , B : Integer ;

begin
Get ( A ) ; Get ( B ) ;
loop
     E := A < B ;
     exit when E ;
     Get ( A ) ; Get (B ) ;
end loop ;
A := B * C - 10 ;
return ;
end P1 ;

function F ( X , Y : Integer ) return Integer is
Val : Integer ;

begin
     if X < Y or X = Y then return X - Y ;
     else return X + Y ;
     end if ;
end F ;
```

```
  procedure P2 ( U , V , W: in out Integer ) is

  begin
       A:= F ( U , 2*V ) ;
       C := A * W ;
      return ;
  end P2 ;
  begin
       Get ( A ) ; Get ( B ) ; Get ( C ) ;
       P2 ( A , B , C ) ;
       Put ( A ) ; Put ( C ) ;
       P1 ;
       Put ( A ) ;
  end Clean_Final ;
```

Figure 11. Program used to test student compilers.

Figure 12 shows the translation for function F. The stack marker is four units long. The calling sequence of instructions saved space for the functional value to be returned. It was left on top of the stack.

```
INCS      1
LOAD I    0 -5      --Load the actual for X
LOAD I    0 -4      --Load the actual for Y
LES  I              --Evaluate
LOAD I    0 -5      -- X < Y
LOAD I    0 -4      -- OR
EQU  I              -- X = Y
OR
FJP  L6             -- No? jump to L6
LOAD I    0 -5      --Yes?
LOAD I    0 -4      --Calculate
SUB  I              --X - Y
STOR I    0 -6      --and return it
UJP L7
UJP L5
LOC L6
LOAD I    0 -5      --Evaluate
LOAD I    0 -4      --X + Y
ADD  I
STOR I    0 -6      --and return it
UJP L7
LOC L5
LOC L7
EXIT 2              --return and pop the stack
```

Figure 12. P-code translation of Function F.

As we made no attempt to optimize the output from the compiler, the code has obvious places where optimizations could be made. Time did not permit us to do that in a beginning one semester course.

```
LOAD @ 0 1
LOAD @ 0 3
LOAD @ 0 2
CLP  3
```

Figure 13. P-code translation of P2 (A , B , C) ;.

This translation places the addresses of the parameters on the stack. The addresses were needed to implement the in out parameter mode. After the addresses are on the stack, the control of the P-code machine passes to the procedure P2.

E. Conclusions and Recommendations

This subset of Ada worked satisfactorily in the course. It provided a suitable project for most of the students. It also provided an opportunity for the students to learn a modest part of the structure of Ada. However, as stated before, the author is of the opinion that except in courses on software engineering only subsets of Ada are feasible in undergraduate courses. He suggests that the DOD requirement of no subsets of the language will have to be relaxed somewhat before the language will find widespread acceptance at the undergraduate level. There needs to be a 'School_Ada' standard; one that will permit '100 dollar compilers' to be written. Then the language will find its undergraduate following. He also suggests that this is the only way that DOD will find the pool of highly qualified people needed to support its Ada programming requirements.

Acknowledgments

Without the help of his wonderful wife, Marian, the author would find it impossible to write papers such as this one. The fine graphics in the paper are the work of his good friend and colleague James R. Brock, Assistant Professor of Engineering, Mesa College. He would also like to thank T. Reising and M. Cummings, two fine students, for the translations shown in Figures 12 and 13.

References and Literature Cited

Barrett, William A., et al., 1986, Compiler Construction Theory and Practice, Second Edition, Science Research Associates, Inc, Chicago, Ill.

United States Department of Defense, Reference Manual for the ADA Programming Language, ANSI/Mil-Std-1815A-1983.

The Use of Ada in the Teaching of Data Structures

Thomas B. Hilburn
Department of Mathematics and Physical Sciences
Embry-Riddle Aeronautical University
Daytona Beach, FL 31014

Abstract

This paper reports the experience of the author in the use of Ada in the teaching of a data structures course to computer science majors at the sophmore/junior level. The software (the Ada compiler and the software support environment) and hardware used in the course are described and their value is assessed. The background of the students and their ability to learn Ada, as a second language, with little formal instruction is described. Finally, the advantages of using Ada in teaching data structure concepts is discussed along with the influence that it has in fostering discussions of the principles of software engineering.

Introduction

In the fall of 1987 and the spring of 1988 the author taught a course in the fundamentals of data structures. The course is a required course in the B.S. degree in Computer Science at Embry-Riddle Aeronautical University. The course is normally taken in the student's sophmore year after the student has taken the courses: Programming I, Programming II, and Introduction to Discrete Structures. The programming courses use Pascal and conform with the features described in recommended revisions of ACM Curriculum 78 [4,5]. The course covered typical topics presented in an introductory data structures course: algorithm analysis, arrays, linked lists, queues and stacks, trees, graphs, sorting and searching, and hash tables. The course text material included handouts and a data structures textbook that uses Ada [3].

Ada was chosen as the programming language for the course because of its emphasis on and support of abstraction and reusability. The expectation of the widespread use of Ada in aviation and aerospace applications was also an important consideration in the selection. (Many Embry-Riddle graduates go on to work in the aviation/aerospace industry). It was decided to use Ada for two semesters, fall of 1987 and spring of 1988, and evaluate its effectiveness and determine the desirability of its future use.

Software/Hardware Environment

The course was serviced by five IBM AT's, each outfitted with a 20 MB hard disk, a single floppy disk drive, and a color monitor. Each AT was installed with an Alsys Ada Compiler and was provided with an editor developed from Borland's Editor Toolbox. The Alsys compiler was part of an Alsys software environment that included utilities for managing Ada libraries and preparing run-time modules. No special utility packages, such as a math functions package or a string handling package, were provided.

Each student maintained their Ada source programs and an Ada library of program units on a floppy disk. Because of the number of students sharing the use of the ATs, students in the data structures course were discouraged from storing their Ada code or maintaining their individual libraries on the hard disks. This allowed the preparation of source code on a non-AT machine (PC or XT) and, similarly, the running of executable modules on a non-AT machine, a considerable saving of limited AT resources.

The resources provided were adequate but not ideal for the purposes of the course. The use of Ada in academe would be greatly improved by a specially designed academic Ada environment. An Ada system (possibly based on a subset of Ada) that would decrease program preparation times could provide a big improvement in Ada education. Large compilation times and extended times for binding and linking are serious impediments to the learning process. Small turn-around times in correcting programming errors are critical to the knowledge acquisition and experimentation that needs to accompany software development in a laboratory setting. One feature of an Ada environment that would reduce program preparation time would be the provision for an Ada-oriented editor that is integrated into the Ada development system (something similar to the editor that is part of Borland's Turbo Pascal package). Also standard utility packages that provide for such tools as string operations, math functions, and set operations would also improve the educational process.

Student Ability to Learn Ada

All students enrolled in the course were sophmores or juniors, had taken at least two programming courses, and all had a working knowledge of Pascal. There was no required Ada text but some of the students obtained copies of [1] or [2]. In addition to Ada, a structured pseudocode was used to describe various algorithms discussed in class. Handouts and special lectures on certain features of Ada were provided when needed. Also, the course textbook had all algorithms written in Ada and it included many explanatory remarks about Ada syntax. There were seven programming projects assigned during the course and after the completion of the first two projects there were minimal problems with Ada syntax.

The similarities between Pascal and Ada syntax made the transition to this new language relatively easy. However, the most useful component of the students' backgr nd which eased their conversion to Ada was the emphasis in their introductory courses on structured programming, modular development and top-down design. The fact that Ada incorporates these features in its design made the learning of Ada seem a natural progression from "theory" to "practice". The emphasis in Ada on packages along with their specification feature was immediately appreciated by students and their comments were in the spirit of "Why don't all languages do it this way?"

With the exception of a few major features (such as generic program units and tasking), most aspects of Ada were used in programming assignments or in examples discussed in class. Features such as private types, unconstrained arrays, overloading of operators, attribute operators, and exception handling were introduced in the discussion of various data structure problems and subsequently were employed by students in programming assignments. To the instructor's surprise, students had little difficulty integrating these concepts into their set of programming tools. Again, upon reflection, the author believes this is due to the combination of the students' background and Ada's design rationale which emphasizes the logical and efficient development of software.

Another pleasant discovery about the teaching of Ada to undergraduate students was the advantages gained in using a package specification to make a programming assignment. A recurring complaint (justified or not) is that the statement of a programming assignment does not provide the student with sufficient information to solve the given problem. Several programming assignments in this data structures course included the specifications of packages to be developed. There were no complaints about vagueness or confusion in the problem statements for these assignments. Of course, other assignments required the student to design their own package specifications; but again the specification feature proved to be a useful pedagogical tool because it provided the student with a mechanism to delineate the essential elements of the solution to a problem (or some part of the problem).

Ada, Data Structures, and Software Engineering

Our study of data structures consisted primarily of the following:

a. definitions of various abstract data types (ADT's),
b. implementation of the ADT's as data structures in a programming environment, and
c. the use of the data structures in application programs.

Ada's package design is a perfect model for demonstrating the relationship between an ADT and a corresponding data structure: the package specification specifies the ADT, a data type and its operations, and a package body contains the executable code necessary to implement the ADT. This parallelism between the theory of the development of data structures and Ada's design philosophy makes quite an impression on the novice programmer and is a powerful teaching tool.

As various features of Ada were developed as part of the discussion of data structures an unanticipated augmentation to the course material took place. During the process of explaining some new element of Ada and the ensuing class discussion, the instructor would find himself engaging in instruction or elaboration of some principle of software engineering. For example, in the discussion of arrays, a matrix package involving unconstrained arrays, overloading of operators, and the use of attribute operators was developed. The use of these Ada features prompted consideration of how they supported the principles of abstraction and reusability in developing software systems. In another example, in an application of linked lists involving a table of student records, the concept of private types was introduced. This naturally led to a discussion of information hiding and its importance in the development of large complex software systems.

Anyone that has used Ada in the development of a software system (even of modest size) is struck by its support of the principles of modularity and localization. The students in the data structures course had previous experience with the modular design of programs in their early courses, but Ada's emphasis on the package concept carries home the benefits and advantages of this concept more succintly and dramatically than any exhortation by a teacher or software guru. An explanation of the need for instantiation of input/output procedures prompts a discussion of the package TEXT_IO; explanation of error messages leads to a discussion of the package IO_EXCEPTIONS; and the use of the operator overloading feature results in a discussion of the STANDARD package. This consideration of several of Ada's predefined packages not only explains some of the details of the Ada architecture it gives the student a working example of a large complex software system that incorporates and uses the principles of modularity and localization.

Conclusions

Ada proved to be an excellent choice for teaching data structures. Students with a background in a structured language such as Pascal and familiarity with good program design had a relatively easy time learning to write Ada code. The chief disadvantage of using Ada from the student's viewpoint were the lengthy compile and binding (linking) times and the weakness of the software development environment. Ada served as an ideal vehicle for illustrating the relationship between an abstract data type and a particular implementation in some data structure. An important secondary benefit of using Ada was the exposure to and learning about software engineering concepts that took place. The author feels that it is almost impossible to use Ada in a programming oriented course without addressing the goals and principles of software engineering.

References

1. ANSI/MIL-STD-1815A Reference Manual for the Ada programming Language, Government Printing Office, Washington, D.C., 1983.
2. Booch, G., Software Engineering with Ada, Benjamin/Cummings, 1987.
3. Feldman, M.B., Data Structures with Ada, Reston Publishing, 1985.
4. Koffman, E.B., Miller, P.L., and Wardle, C.E., "Recommended Curriculum for CSI, 1984", Communications of the ACM 27, 10 (Oct. 1984), 998-1001.
5. Koffman, E.B., Miller, P.L., and Wardle, C.E., "Recommended Curriculum for CS2, 1984", Communications of the ACM 28, 8 (Aug. 1985), 815-818.

This Page Left Blank Intentionally

A Coherent Set of Ada Exercises

by

Richard F. Sincovec *
Computer Science Department
University of Colorado
Colorado Springs, Colorado 80933-7150
(719) 593-3332

Abstract

In this paper, we present a set of Ada computer assignments that are designed so that each assignment builds on the results of prior assignments. This set of assignments illustrate most of the important features of Ada including packages, generics, input/output, tasks, exceptions and incorporate data abstraction, information hiding, reusability and concurrency.

Introduction

Ada contains a number of language features that encourages the development of software systems as a collection of reusable software components. The natural way to teach Ada is to begin with assignments that use existing software components, then progress to assignments that require the design and implementation of the previously used, as well as, new components, and, finally, to end with assignments that use the components created in earlier assignments. Each assignment by itself should be simple enough so that the student can grasp the features of Ada without being bogged down in details associated with large assignments. The collection of assignments should be comprehensive with respect to learning most of the Ada language.

In this paper, we present such a coherent set of Ada assignments. These assignments have been successfully used in various Ada classes over the past five years.

----

* Author's present address: Research Institute for Advanced Computer Science, Mail Stop 230-5, NASA Ames Research Center, Moffett Field, CA 94035, (415) 694-6363

The Appendix contains a possible solution to one of the latter assignments. This should be sufficient to derive possible solutions to most of the other assignments because of the dependency of this assignments on the earlier assignments.

Assignment 1: Getting Started

The purpose of this assignment is to gain familiarity with the Ada environment that you will be using by creating and using an Ada program library, instantiating portions of text_io into your Ada program library, and creating, compiling and running a simple Ada program.

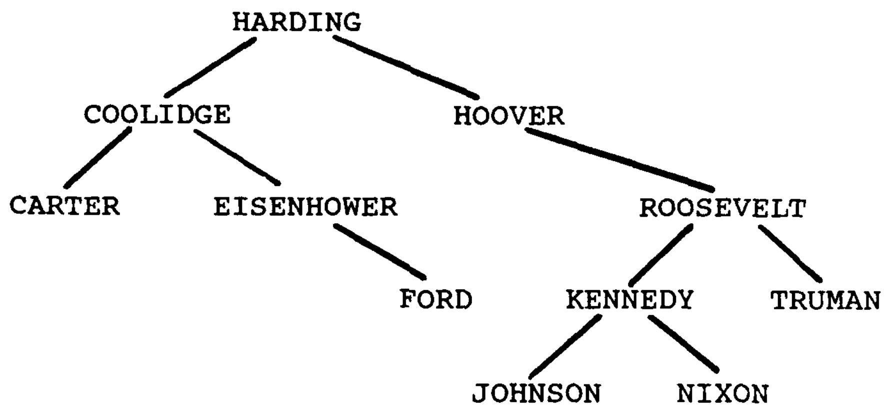
Log onto the system that you will be using. Read the appropriate documentation and create your Ada program library. Use an editor to create a program unit that instantiates integer_io and float_io, compile the unit placing these instantiations into your Ada program library. Now use an editor to create a simple Ada program that illustrates the use of text_io, integer_io and float_io. Your simple Ada program should also use string_io and character_io. Finally, in your simple program define an enumerated type, instantiate enumeration_io for the type you defined and illustrate the use of enumeration_io. Compile, link, and execute your simple Ada program. Use some of the utilities that are available in your Ada environment to maintain and manipulate your Ada program library.

Background information for the remaining assignments.

Binary trees have a variety of uses. A binary search tree can be used to maintain an ordered collection of elements provided the tree is constructed to satisfy the following search tree property. For each node of a binary search tree, its subtree is defined according to the element in the node. If a left subtree is present, it contains elements that are smaller. If a right subtree is present, it contains elements that are greater.

Consider the following example where the elements are strings:

Input:
HARDING
COOLIDGE
HOOVER
ROOSEVELT
TRUMAN
EISENHOWER
KENNEDY
JOHNSON
NIXON
FORD
CARTER



To determine whether an element is present in a binary search tree, we use the properties of search trees to simplify the search. Instead of examining every node, we start at the root and take either the left or right subtree at each stage. If the node contains an element larger than the search element, we go left, if smaller, we go right, otherwise, that's it. This can be implemented using a recursive function.

When elements are inserted into a binary tree, we insert them as leaf nodes such that the search tree property is maintained. The algorithm uses recursion.

The process of visiting every node in a binary search tree is called traversal. To traverse a binary search tree in the relational order of its elements, we follow three steps at each node starting with the root node. First, we traverse its left subtree if present, next we "visit" the node itself, and, finally, we traverse its right subtree. Recursion again.

You are initially provided with an Ada package as follows for binary search trees with string elements:

```
package binary_search_tree is

  type tree is limited private;

  procedure insert
          (  t  : in out tree;
             e  : in     string );
  -- Insert a string, e, into the tree, t.
  procedure print
         ( t  : in tree );
  -- Traverses each node in the tree printing its contents.

  function in_tree
         ( t  : in tree;
           e  : in string ) return boolean;
  -- Returns true if the string, e, is in the tree, t,
  -- otherwise, returns false.

private

  type tree_node;
  type tree is access tree_node;
  type tree_node is
    record
      item  : string(1..20);
      left  : tree;
      right : tree;
    end record;

end binary_search_tree;
```

Assignment 2: Interfacing to Existing Packages

The purpose of this assignment is to illustrate the use of reusable software component based on data abstraction and information hiding, the use of string_io, and the use of simple control structures.

Write a simple driver program that uses an existing binary tree handler package. Get a string from the console and insert it into the binary tree. This process is repeated until a sentinel string (value) is entered. Check to be sure that the input value is not already in the search table before you insert it. Finally, traverse the tree printing each element. The output should be a listing of the input data in alphabetical order.

Assignment 3: File Input/Output

The purpose of this assignment is to become familiar with text file input/output, procedures, and separate compilation.

Create an input text file using a text editor. Design and implement a procedure (get_word) that reads one word at a time from a text file. The file may contain characters other than letters. A word consists of upper and lower case characters and the underline character. For words over 20 characters in length, the excess characters are ignored. Use the functions and procedures available in text_io. Separately compile your get_word procedure into your Ada program library. In anticipation of later assignments, have your get_word procedure return the line number in which the word appears in the input file. A function in text_io returns the line number.

Now write a main program that uses your get_word procedure and the binary search tree package. Read words from the input text file. A good input file might be the program that you create for this assignment. Place each distinct word in a binary tree using the operations provided in the binary tree package. When the end of the input text file is reached, traverse the tree and write each word to an output file by using the redirection features of text_io. The output file contains all the distinct words in the input file in alphabetical order.

Assignment 4: Spelling Checker

The purpose of this assignment is to introduce exception handling, subunits and to illustrate the use of reusable software components including the binary search tree package and the previously developed get_word procedure.

Create a text file to be spell checked and a text file that contains the words in your dictionary. The name of the text file containing dictionary words is requested and the contents of the file read to form a binary search tree which contains all the words in the dictionary. The creation of the dictionary tree should be done using a procedure whose body should be stubbed out. Each word in the text file that is to be spell checked is read and compared with the words in the dictionary. Misspelled words are written to an output file. The user corrects misspelled words by examining the output file and editing the input file. The name of the text file to be spell checked is requested user input. Exception handlers should be designed to handle any input errors by the user.

An extension of this assignment is to develop an interactive man machine interface to handle misspelled words as they are encountered and to correct them in the input text.

Assignment 5: Queue Package Implementation

The purpose of this assignment is to become familiar with Ada packages by implementing an abstract data type.

Implement the package body for the queue abstract data type for queues of integers. The queue should be implemented as a circular queue. Write a menu driven test program that uses local exception handlers to capture any user errors, inform the user of the error, and continue execution. The test program should test all operations and exception conditions of the queue package.

The queue package specification is given below:

```
package queue_adt is

   -- Package for the queue abstract data type.  The size
   -- of the queue is specified when declared in the client
   -- program.  The queue is initialized when it is declared.

   type queue( size : natural ) is private;
   -- The queue may be declared as limited private since the
   -- operations of checks for equality and inequality are
   -- not valid.

   procedure make_empty( q : in out queue );
   -- Reinitializes the queue to the empty queue.

   function is_empty( q : queue ) return boolean;
   -- Returns true if queue is empty, otherwise, returns false.

   procedure insert( q : in out queue; item : in integer );
   -- Inserts an item at the rear of the queue.

   procedure remove( q : in out queue; item : out integer );
   -- Removes an item from the front of the queue.

   function length( q : queue ) return natural;
   -- Returns the length of the queue.

   underflow : exception;
   overflow  : exception;
   -- Underflow is raised by remove if attempting to remove an
   -- item from an empty queue.  Overflow is raised by insert
   -- if attempting to insert an item into a full queue.  A
   -- queue is full if the number of items in the queue is equal
   -- to its declared length.
```

```
private
  type storage is array( integer range <> ) of integer;
  type queue( size : natural ) is
    record
      items  : storage( 0..size );
      front  : integer := 0;
      rear   : integer := 0;
      length : natural := 0;
    end record;
end queue_adt;
```

Assignment 6: Search Tree Package Implementation

The purpose of this assignment is to implement a second abstract data type that requires the use of the dynamic allocation features of Ada and recursion.

Implement the package body for the binary search tree. The package specification is given above. Test your implementation using your programs from Assignments 2, 3 and/or 4.

Depending on the class, this assignment may be skipped and the source code for the binary search tree package body given to the students as initial information for the next assignment.

Assignment 7: Generic Binary Search Tree Package

The purpose is to illustrate most of the considerations in the design and implementation of a reusable generic software package based on data abstraction.

The problem is to implement the binary search tree package as a generic package with the appropriate generic formal parameters and test it using your programs from Assignment 2, 3 and/or 4. You should also test your program for elements defined as a record data structure. For record elements that contain more than one field, a field of the record is designated as the key field on which an order relationship is defined for creating and searching the tree. Defining and using the appropriate generic parameters is the focus of this assignment. It is not as easy as it sounds!

In anticipation of later assignments, modify the binary search tree specification to include the additional operation that permits an element to be updated and whose interface is given by:

```
procedure update
        (  t  : in out tree;
           e  : in     element );
-- Update the element, e, in the tree, t, using the function
-- equal to find the element, e.  The field of e that specifies
-- the order relationship for the elements in the tree (the key
-- field) should not be modified.  Update is only applicable
-- to trees that contain multiple field elements.
```

The generic parameters are:

element	- The type of elements in the tree. For record elements, one field of the record is the key field on which the equal ("=") and "<" functions are defined.
put_element	- Procedure specifying the operations to perform when each element in the tree is visited by print.
update_element	- Procedure specifying the operations to perform when updating an element in the tree. The key field should not be modified.
equal or "="	- Function specifying the meaning of equality for two elements in the tree. Based on the key field of the elements.
"<"	- Function specifying the order relationship for two elements in the tree. Based on the key field of the elements.

Assignment 8: Cross Reference Generator

The purpose of this assignment is to illustrate the layering of abstractions.

In this assignment, you will use both the queue and the binary search tree abstract data types. You will also use the get_word procedure that you wrote for a previous assignment. The first thing that you need to do is to make the queue package from Assignment 5 generic with respect to the type of elements in the queue. This is easy.

A cross-reference generator reads a source text and prints a list consisting of one entry for each distinct word in the text. Each entry consists of the word itself followed by a list of the numbers of the lines on which it appears. Letters after the twentieth letter of a word may be ignored. The words are listed in alphabetical order. If a word occurs n times on one line, the line number should be printed n times. The output must be formatted for pages of 60 lines and no line may contain more than 80 characters. Use the format control subprograms in text_io to format the output.

Each node in the binary search tree contains a record consisting of two fields: a word and a queue of line numbers on which the word was found in an input text file. Words are read from the input text file until the end of the file is encountered. If the word is not in the binary search tree, then it is added to the tree and the line number on which the word occurred is inserted into the queue. If the word is in the tree, then the appropriate node in the search tree is updated by inserting the line number into the queue associated with that word.

Assignment 9: Concurrent Spelling Checker/Input Editor

The purpose of this assignment is to illustrate and use the tasking features of Ada. This program utilizes the get_word procedure, the generic queue abstract data type from Assignment 8 and the generic binary search tree abstract data type from Assignment 7.

Concurrently, as words are input from the terminal they are checked to see if they are in a dictionary which is stored as a binary search tree. If a typed word is misspelled the terminal bell rings and the misspelled word is written to an output file of misspelled words. The output is two files: one contains the misspelled words and the other contains the input file.

The implementation involves several communicating tasks. A producer task performs the editor functions and a consumer task performs the spelling checker functions. Since these two tasks proceed at their own rates, a control task should be defined to buffer the data which passes between them. As each new word is input to the producer task, it is sent to the control task where it is picked up by the consumer task. If the word is not in the dictionary, then the consumer task rings the terminal bell and writes the potentially misspelled word to an output file. The control task needs to buffer a queue of words.

The main difficulty is the implementation of a character input interrupt. That is, as each character is typed it must be read immediately and processed. Some Ada environments provide such a function. Text_io requires a line feed to read the characters input from the keyboard.

Conclusions

In this paper we have presented a coherent set of Ada exercises that can be used in an Ada course to illustrate most of the features of the Ada language. Students have found these exercises interesting, as well as, challenging and have emerged from the course with a comprehensive knowledge of the Ada language.

Appendix

This appendix contains a possible solution to Assignment 8.

```
with text_io; use text_io;
with integer_text_io; use integer_text_io;
with queue_adt;
with generic_binary_search_tree;
with get_word;
procedure cross_reference_generator is

-- Cross reference generator.
--
-- Written by Richard F. Sincovec
--             November 6, 1987
--
-- This program reads a text file and prints a list consisting
-- of one entry for each distinct word in the text.  Each
-- entry is followed by a list of numbers of the lines on
-- which the word appears.  A word is a string of consecutive
-- lowercase, uppercase or underline characters.  Letters
-- after the 20th are ignored.  The words are listed in
-- alphabetical order.

package line_number_queue is new queue_adt( positive_count );
use line_number_queue;
-- A queue is used to store the line numbers on which a word
-- appears.  The type positive_count is from text_io.

subtype word is string(1..20);
-- Defines word.

type word_record is
  record
    w : word;
    q : queue( 50 );
  end record;
```

```
-- Defines information needed for each word: the word and a
-- queue containing the line numbers in the text file where
-- the word appears.  The queue size is arbitrarily set at
-- 50.  The word field of the record is the key field for use
-- with the binary search tree.

procedure put_word_record( wr : in word_record );
-- For use with the generic binary search tree.  Specifies
-- how to output a node in the tree.

procedure update_word_record( wr : in out word_record );
-- For use with the generic binary search tree.  Specifies
-- how to update a node in the tree.  It adds a line number
-- to the queue associated with the node.

function "<"( wr1, wr2 : in word_record ) return boolean;
-- For use with the generic binary search tree.  Defines the
-- meaning of less than for two nodes.

function equal( wr1, wr2 : in word_record ) return boolean;
-- For use with the generic binary search tree.  Defines the
-- meaning of equality for two nodes.

package word_handler is new generic_binary_search_tree
      ( element        => word_record,
        put_element    => put_word_record,
        update_element => update_word_record,
        equal          => equal,
        "<"            => "<" );
use  word_handler;
-- A binary search tree is used to store word records.

-- Variable declarations.
fd         : text_io.file_type;
file_name  : string(1..32);
length     : natural;
node       : word_record;
line_no    : positive_count;
t          : tree;

procedure put_word_record( wr : in  word_record ) is
  wr_local    : word_record := wr;
  line_number : positive_count;
begin
  -- Output the word.
  put( wr_local.w );
  -- Output the contents of the queue.
  while not is_empty( wr_local.q ) loop
    remove( wr_local.q, line_number );
    -- If the line is the right margin, then skip to a
    -- new line.
    if col > 76 then
```

```
        new_line;
        set_col( 21 );
      end if;
      put( integer(line_number), 4 );
    end loop;
    new_line;
  end put_word_record;

  procedure update_word_record( wr : in out word_record ) is
  begin
    -- Insert line number into the queue.
    insert( wr.q, line_no );
  end update_word_record;

  function "<"( wr1, wr2 : in word_record ) return boolean is
  begin
    return wr1.w < wr2.w;
  end "<";

  function equal( wr1, wr2 : in word_record ) return boolean is
  begin
    return wr1.w = wr2.w;
  end equal;

  begin  -- Cross reference generator
    -- Get file name and open file.
    put( "Enter name of file for word concordance --> " );
    get_line( file_name, length );
    open( fd, in_file, file_name(1..length) );

    -- Read words in file and build tree.
    while not end_of_file( fd ) loop
      get_word( fd, node.w, line_no );
      if not in_tree( t, node ) then
        make_empty( node.q );
        insert( node.q, line_no );
        insert( t, node );
      else
        update( t, node );
      end if;
    end loop;
    close( fd );
    new_line;

    -- Traverse all elements in the tree and print contents of
    -- each element.
    print( t );
    new_line;

  end cross_reference_generator;

  -------------------------------------------------------------
```

```
with text_io; use text_io;
procedure get_word( f : file_type; word: out string;
                    line_number : out positive_count ) is

  -- Gets a word from the source text file.  A word consists
  -- of upper and lower case characters and the underline
  -- character.  This could be easily modified to include
  -- numerical characters.

  -- For words over 20 characters in length, the excess
  -- characters are ignored.  A blank word is returned if
  -- there are no more words in the file.  This procedure may
  -- be separately compiled.

  char       : character;
  length     : natural := 0;
  max_length : constant := 20;

  function is_word_character( char : character ) return
           boolean is
    subtype lower_case is character range 'a'..'z';
    subtype upper_case is character range 'A'..'Z';
  begin
    return ( char in lower_case ) or ( char in upper_case )
           or ( char = ascii.underline );
  end is_word_character;

begin
  word := (1..max_length => ' ');
  while not end_of_file( f ) loop
    while not end_of_line( f ) loop
      get( f, char );
      if is_word_character( char ) then
        if length < max_length then
          length := natural'succ( length );
          word( length ) := char;
        end if;
      elsif length > 0 then
        line_number := line( f );
        return;
      end if;
    end loop;
    line_number := line( f );
    if not end_of_file( f ) then
      skip_line( f );
    end if;
    exit when length > 0;
  end loop;
end get_word;

-------------------------------------------------------------
```

```
generic
  type element is private;
  with procedure    put_element( e : in     element ) is <>;
  with procedure update_element( e : in out element ) is <>;
  with function equal( e1, e2 : element ) return boolean
                is <>;
  with function  "<" ( e1, e2 : element ) return boolean
                is <>;

package generic_binary_search_tree is

  -- Generic parameters:
  --   element        - The type of elements in the tree.
  --                    For record elements, one field of the
  --                    record is the key field on which the
  --                    equal and "<" functions are defined.
  --   put_element    - Procedure specifying the operations to
  --                    perform when each element in the tree
  --                    is visited by print.
  --   update_element - Procedure specifying the operations to
  --                    perform when updating an element in
  --                    the tree.  The key field should not
  --                    be modified.
  --   equal          - Function specifying the meaning of
  --                    equality for two elements in the tree.
  --   "<"            - Function specifying the order
  --                    relationship for two elements in the
  --                    tree.

  -- Written by Richard F. Sincovec
  --            January 26, 1987

  type tree is limited private;

  procedure insert
        ( t  : in out tree;
          e  : in     element );
  -- Insert an element, e, into the tree, t, using the
  -- function "<" to determine the proper position in the
  -- tree.

  procedure update
        ( t  : in out tree;
          e  : in     element );
  -- Update the element, e, in the tree, t, using the function
  -- equal to find the element, e.  The field of e that
  -- specifies the order relationship for the elements in the
  -- tree (the key field) should not be modified.
  procedure print
          (  t   : in tree );
  -- Traverses each element in the tree calling the user
  -- defined procedure put_element.

  function in_tree
          (  t   : in tree;
             e   : in element ) return boolean;
  -- Returns tree if the element, e, is in the tree, t,
  -- otherwise, returns false.

private

  type tree_node;
  type tree is access tree_node;
  type tree_node is
    record
      item  : element;
      left  : tree;
      right : tree;
    end record;

end generic_binary_search_tree;

-------------------------------------------------------------

package body generic_binary_search_tree is

  -- Implementation of the generic binary search tree
  -- package.
  --          Written by Richard F.  Sincovec
  --          January 26, 1987

  procedure insert
          (  t   : in out tree;
             e   : in     element ) is
  begin
    if t = null then
      t := new tree_node'( e, null, null);
    elsif e < t.item then
      insert( t.left, e );
    else
      insert( t.right, e );
    end if;
  end insert;

  procedure update
          (  t   : in out tree;
             e   : in     element ) is
  begin
    if t /= null then
      if equal( t.item, e ) then
```

```
        update_element( t.item );
      elsif e < t.item then
        update( t.left, e );
      else
        update( t.right, e );
      end if;
    end if;
  end update;

  procedure print
          (  t  : in tree ) is
  begin
    if t /= null then
      print( t.left);
      put_element( t.item );
      print( t.right );
    end if;
  end print;

  function in_tree
          (  t  : in tree;
             e  : in element ) return boolean is
  begin
    if t = null then
      return false;
    elsif equal( e, t.item ) then
      return true;
    elsif e < t.item then
      return in_tree( t.left, e );
    else
      return in_tree( t.right, e );
    end if;
   end in_tree;

end generic_binary_search_tree;

--------------------------------------------------------------

generic
  type element is private;

package queue_adt is

  -- Written by Richard F. Sincovec
  --            January 26, 1987

  -- Generic package for the queue abstract data type.  The
  -- size of the queue is specified when declared in the
  -- client program.  The queue is initialized when it is
  -- declared.

  type queue( size : natural ) is private;
  procedure make_empty( q : in out queue );
  -- Reinitializes the queue to the empty queue.

  function is_empty( q : queue ) return boolean;
  -- Returns true if queue is empty, otherwise, returns false.

  procedure insert( q : in out queue; item : in element );
  -- Inserts an item at the rear of the queue.

  procedure remove( q : in out queue; item : out element );
  -- Removes an item from the front of the queue.

  function length( q : queue ) return natural;
  -- Returns the length of the queue.

  underflow : exception;
  overflow  : exception;
  -- Underflow is raised by remove if attempting to remove an
  -- item from an empty queue.  Overflow is raised by insert
  -- if attempting to insert an item into a full queue.  A
  -- queue is full if the number of items in the queue is
  -- equal to its declared length.

private
  type storage is array( integer range <> ) of element;
  type queue( size : natural ) is
    record
      items  : storage( 0..size );
      front  : integer := 0;
      rear   : integer := 0;
      length : natural := 0;
    end record;
end queue_adt;

-------------------------------------------------------------

package body queue_adt is

  -- Implementation of the queue abstract data type.
  -- Written by Richard F. Sincovec
  --           January 26, 1987

  procedure make_empty( q : in out queue ) is
  begin
    q.front := 0;
    q.rear  := 0;
    q.length:= 0;
  end make_empty;
  function is_empty( q : queue ) return boolean is
  begin
    return q.front = q.rear;
  end is_empty;

  procedure insert( q : in out queue; item : in element ) is
  begin
    if q.front = (q.rear + 1) mod q.items'length then
      raise overflow;
    else
      q.rear := ( q.rear + 1 ) mod q.items'length;
      q.items( q.rear ) := item;
      q.length := q.length + 1;
    end if;
  end insert;

  procedure remove( q : in out queue; item : out element ) is
  begin
    if q.front = q.rear then
      raise underflow;
    else
      q.front := ( q.front + 1 ) mod q.items'length;
      item := q.items( q.front );
      q.length := q.length - 1;
    end if;
  end remove;

  function length( q : queue ) return natural is
  begin
    return q.length;
 end length;


end queue_adt;
```

Previously published in Proceedings of the Software Engineering Education Conference
Washington, D.C., April 28-29, 1988

The Software Engineering Graduate Program at the Boston University College of Engineering

John Brackett, Thomas Kincaid and Richard Vidale
Department of Electrical, Computer and Systems Engineering
College of Engineering, Boston University

The College of Engineering has developed a Master's degree program in software engineering to meet the needs of industrial software development and management. The program will educate software engineers by providing courses in the technology, methodology and management of software development. The program incorporates the best features of the Master of Software Engineering curriculum formerly offered at the Wang Institute of Graduate Studies [1,2] and the MS in Systems Engineering, Software Engineering Option, offered at Boston University. A doctoral program leading to the Ph.D. in Engineering, with research specialization in software engineering, is also available in the College of Engineering.

The software engineering Master's program is offered in the Department of Electrical, Computer and Systems Engineering as the Software Engineering Option of Master of Science in Systems Engineering; the program is expected to be renamed the Master of Science in Software Systems Engineering by the fall of 1989. The program emphasizes the understanding of both hardware and software issues in the design and implementation of software systems. Special emphasis is placed on the software engineering of two important classes of computer systems: embedded systems and networked systems. Ada is the language used in a majority of the courses.

Both full- and part-time programs are available, and a majority of the program will be available on television for those corporate locations which are members of the Boston University Corporate Classroom interactive television system. The program may be completed in twelve months by full-time students.

I. Curriculum

The master's program requires the completion of nine four-credit semester-length courses: six required courses, two technical electives, and a team project. There are two entrance tracks: one for those with a hardware background (Electrical or Computer Engineering) and another for those with a software background (Computer Science or work experience in software development).

The required courses common to both tracks are:

- Applications of Formal Methods
- Software Project Management
- Software System Design
- Computer as System Component
- Software Engineering Project

Those students with a hardware-oriented background take the sequence:

> Advanced Data Structures
> Operating Systems

Those students with a software-oriented background take the sequence

> Switching Theory and Logic Design
> Computer Architecture

The objective of the two sequences is to provide each group of students with material lacking in their background. The computer engineering program in the College of Engineering provides the basis for the hardware-oriented aspects of the program.

The prerequisite structure of the program is shown in Figures 1 and 2; the catalog description of the courses are included in the Appendix. The technical electives may be selected from courses in the areas of software engineering, computer engineering and computer science.

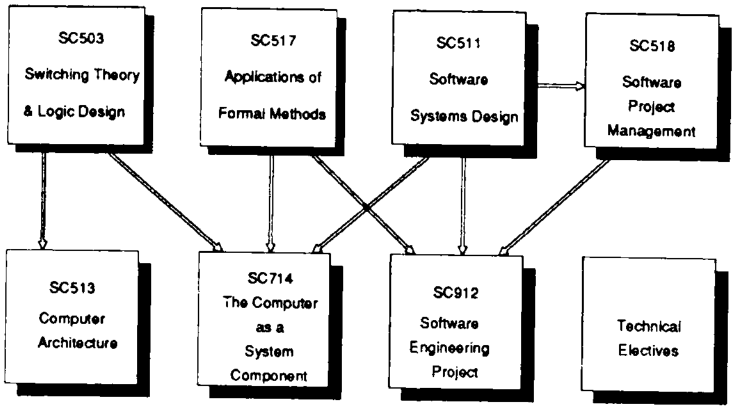

SC503 Switching Theory & Logic Design
SC517 Applications of Formal Methods
SC511 Software Systems Design
SC518 Software Project Management
SC513 Computer Architecture
SC714 The Computer as a System Component
SC912 Software Engineering Project
Technical Electives

Figure 1: Prerequisite Structure, Software Background

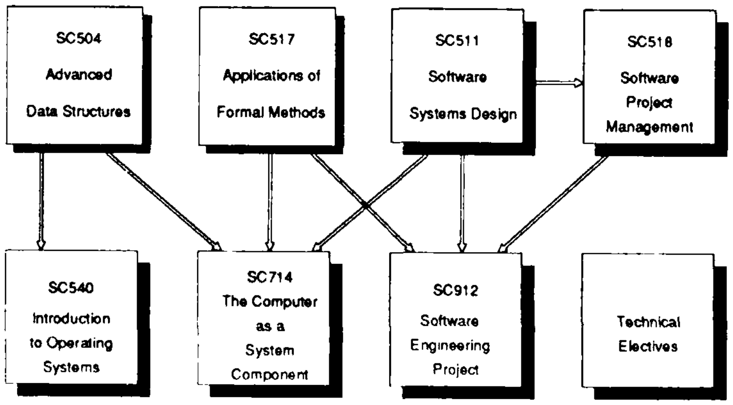

SC504 Advanced Data Structures
SC517 Applications of Formal Methods
SC511 Software Systems Design
SC518 Software Project Management
SC540 Introduction to Operating Systems
SC714 The Computer as a System Component
SC912 Software Engineering Project
Technical Electives

Figure 2: Prerequisite Structure, Hardware Background

The capstone software development project is performed by teams of students during a single semester. The project is designed to integrate and apply the knowledge acquired in the program, and emphasizes team techniques, communication skills, and planning. Projects are judged by both academic and industrial standards, and are closely supervised by a faculty member.

II. Admission Requirements

The minimum admission requirements to the program are:

BS from accredited institution, with at least a B average
Programming experience in:
 a block structured, high level language
 an assembly language

Courses in discrete mathematics, probability and statistics, and data structures are recommended, but are not required for admission. However, students without this background, which may be gained either through courses or work experience, may be required to complete remedial material or courses before beginning the program. Successful work experience in software development is preferred, but not required.

The College plans to restrict enrollment in the program to 60 full-time equivalent degree candidates. Since many of the students will be part-time students, 125-150 students are expected to be masters degree candidates by the 1990-91 academic year. The enrollment in the degree program in the 1988 spring semester was approximately 20, plus those students who were taking courses in the program as a non degree candidates. Many of these special students are expected to become degree candidates.

III. Cooperation with Industry

The College of Engineering has given a high priority to the strengthening of ties with companies involved in software development. The curriculum has been designed, with significant industrial input, to meet the needs of industry for students trained in both the technical and management aspects of software engineering. The College has formed an Industrial Advisory Board, comprised of technical leaders from major Boston-area companies, to advise the Dean and the faculty on the program's development.

In the Boston area there is a high demand for software engineers experienced in the development of software for computers which are components of larger systems. Although many of the job openings involve embedded systems for the Department of Defense, there are also many companies developing computer-based products for a diverse range of commercial applications. Therefore, the program is designed to educate software engineers who can develop high-quality software that works closely with a product's hardware.

In addition to the professional Masters program, the College is expanding its software-oriented research activities. Ph.D. level research in software engineering is expected to become an important part of the College's research activities during the next few years. The University recognizes that rapid advances in software engineering during the last decade have been spurred by research conducted in both academic and industrial settings. Therefore an important goal of the College is to forge effective technology transfer paths between the academic and industrial software research communities.

IV. Television-Based Instruction

In response to industrial requests, Boston University has given high priority to meeting the needs of students who can take only one or two courses a semester. The College of Engineering developed the Corporate Classroom interactive television system, beginning in 1983, in order to meet the needs of part-time students that can not easily commute to the main Boston University campus.

Courses are transmitted live from classrooms on campus. Off-campus students may ask questions by telephone. Student questions in the classroom can be heard by students in the television audience, and telephone questions are amplified so they can be heard in the classroom. Nearly all the courses are broadcast during the day, and most companies participating in the program give employees release time to view classes. Video tapes are available if a student misses a class or for review purposes. A courier takes materials to each corporate site and collects homework and project deliverables from the students. A student can attend on-campus classes, and instructors are available during specific hours by telephone for individual consultation. All exams are taken with the on-campus students. Many faculty meet personally with the students to review project work.

Five of the six required classroom courses in the software engineering masters program will be telecast; the exception is "The Computer as System Component" which will require use of the on-campus Embedded System Laboratory. In addition, a selection of elective courses will be available on television. Therefore, a student who carefully selects electives may complete seven of the nine courses required for the degree on television, and will need to come to campus for only two courses, the project course and "The Computer as a System Component."

Interest in taking the software engineering program on television is surprisingly high among students with management responsibility. For example, two of the best students in one of the authors' software requirements definition class in the fall of 1987 were managers with 35 and 100 people in their organization. These managers said television was the only way they could have taken the course, since their responsibilities made travel to the campus impossible. Both of them used video tapes extensively to make up for classes missed during business travel. When they did watch the class live, they asked some of the most penetrating questions.

Some important issues related to teaching software engineering on television have yet to be resolved. Students must be encouraged to form teams to work on projects, even if it means traveling to another site. Because some students are from locations where visitors are not allowed outside of normal working hours, team project work is hard to arrange. However, we believe it is a vital part of the program. One possible approach, as yet untried, is to provide meeting and computer facilities at least one Boston University suburban campus location.

The most important issue to be resolved is how to provide students with state of the art software engineering tools to support their course work. The television courses have traditionally relied upon dial-up access to computers on the main campus. However, with the increased importance of graphically-oriented tools on powerful personal computers and workstations, the software engineering program should not restrict the tools used in courses to those that are available via character-oriented terminals on 1200 baud telephone lines. The most obvious solution would be to request the companies sponsoring students to provide a "standard" workstation and a set of specified software tools. Since some of the sponsor companies manufacture workstations, this solution is not practical. The most feasible solution appears be to specify a set of software tools that are available on several hardware platforms and request that the sponsoring companies provide them to their students. We expect the members of the Industrial Advisory Board will need to focus attention in their organizations on how to have state of the art tools readily available to their students.

V. Program History

The Software Engineering Option in the Systems Engineering Master's program was approved by the College of Engineering faculty in the spring of 1980. The first degree was granted in 1982. The program enjoyed a slow, but steady, growth until sixteen students were matriculated in the spring of 1987. Following the acquisition of the Wang Institute by Boston University, Digital Equipment Corporation requested that the university seriously consider a major expansion of the existing College of Engineering program. During the early summer of 1987 the College faculty and administration prepared a proposal for the companies that had sponsored students at the Wang Institute. In July 1987 DEC agreed to provide a leadership grant to expedite the establishment of the expanded program. The acquisition of the software engineering library of the Wang Institute has given the College's students and faculty access to an excellent collection of software engineering materials.

The revised curriculum was approved by the College faculty in October 1987 and introduced in January 1988; it retains the original program emphasis on understanding both hardware and software issues in the design and implementation of software systems. The revised program incorporates major elements of the Master of Software Engineering curriculum formerly offered at the Wang Institute. 4 of the 8 required courses were presented in the spring of 1988; all of the required courses will be offered during the 1988-89 academic year.

VII. Comparison with the Wang Institute MSE Program

It is impossible to briefly compare the Wang Institute Master of Software Engineering program with the current College of Engineering program. However, it is useful to state three major differences between the two programs:

1. The WI program was oriented toward the software engineering of system software, such as operating systems and compilers, and stand alone software products. The BU program is oriented toward software systems that are an integral component of a larger system.

2. The WI program was designed for full-time students and part-time students who could obtain company release time to travel to the Tyngsboro campus during the work day. The BU program uses television to reach part-time students in their company and over a wider geographical area than was served by the WI.

3. The WI program required a minimum of one year of related work experience; the BU program has no work experience requirement. However, we expect the work experience of degree candidates to be, on the average, approximately equal.

VIII. Future Challenges and Directions

The limiting factor in the growth of the program is the ability to recruit qualified faculty. Recruiting faculty with the required academic credentials and experience in software engineering is very difficult, and the emphasis in the program on hardware and systems issues makes the job even harder. We anticipate that some of the additional faculty will have a computer engineering background, with significant experience in software development.

A significant number of full-time Masters and Ph.D. students is essential to a high quality graduate program. These students, many of whom will be company sponsored and able to attend any one of the top research universities, must be actively recruited.

The corporate supporters of the program are already requesting that our courses be available by television to sites outside the current 40 mile signal range. Live satellite transmission of our courses is obviously feasible, and the College may test it at pilot sites during the 1988-89 academic year. However, an important part of the pilot program will be to investigate techniques, such as two-way video conferencing, that will allow a faculty member to review student project work remotely.

During the past year the College of Engineering has taken major steps to expand its activities in software engineering education. The current College of Engineering program owes a great deal to the pioneering work in software engineering education done at the Wang Institute. The curriculum developed at the Wang Institute has greatly influenced the content of the revised program. In addition, the 130 Wang Institute MSE graduates have demonstrated to their companies the value of graduate education in the field. Many of these graduates have been strong advocates of financial support from industry to preserve a software engineering program in the Boston area.

Dr. Wang recognized the need for industrially-oriented, high quality graduate education in software engineering long before most of his peers. Now that there is a wider perception of the need, we are confident that a long term university-industrial partnership can build a nationally recognized graduate program in software engineering at Boston University.

Bibliography

[1] M. Ardis, *Evolution of the Wang Institute Master of Software Engineering Program.* IEEE Transactions on Software Engineering, November, 1987.

[2] S. Gerhart, *Skills versus Knowledge in Software Engineering Education: A Retrospective on the Wang Institute MSE Program.* in Software Engineering Education: The Educational Needs of the Software Community, Springer-Verlag, 1987.

Advanced Learning Technologies Project
Software Engineering Institute
Carnegie Mellon University

Ada Software Engineering Education & Training Symposium

May 1, 1988

Scott M. Stevens
Michael Christel
Judy Chiswell

This work sponsored by the Department of Defense

Table of Contents

1. Introduction

The Advanced Learning Technologies Project at the Software Engineering Institute is applying advanced hardware and software technologies to software engineering education. The content domain of the project is formal technical review of Ada code. The specific type of review methodology taught is an inspection. Code inspection is a formal review process that has proven to be an effective technique for defect identification. Appropriately applied, code inspection is also a management tool providing visibility into the software process.

The Advanced Learning Technologies Project has two goals. First, it will illustrate the applicability of technologically advanced educational media to software engineering problems. These media include Interactive Videodisc, intelligent tutors, advice-giving expert systems, Compact Disc-Interactive, and Digital Video Interactive. Second, the project will help to transition effective software engineering methods into practice by providing an actual course.

The National Science Board recognizes the need for improved technical education [NSB 86]:

> The mechanisms and modes of delivery of instruction have taken on significance nearly as great as the content, with the advent of the new technologies, especially the computer. Ways must be sought to exploit the power of these technologies in the learning process, in the interests of increased efficiency and effectiveness of learning and lower overall costs.

Software engineering in particular has been a difficult area for educators. Software engineering continues to be human-intensive, especially in areas such as project management and technical review. Most teaching for skills in these areas depends on group-paced learning and on instructors highly skilled in designing and delivering group-process oriented instruction. As a result, the quality of software engineering education often suffers; instructors are primarily technical people, skilled in and oriented toward software engineering *technical* issues, but not experienced in instructional design or group leadership.

Developing the teaching expertise needed for these courses is difficult. IBM uses software developers and project managers with an average of ten years experience as educators in a two week course on software engineering. Before they are allowed to lead this class, these people are given nine months of full-time training in how to teach this course [Pietrasanta 87]. Obviously, most institutions do not have the resources necessary for this approach. Advanced learning technologies can capture this expertise and deliver it when and where it is needed. These technologies have proven to be fruitful teaching tools as well as cost effective in numerous disciplines. The Advanced Learning Technologies Project demonstrates their efficacy in the field of software engineering through a course on and simulation of an Ada code inspection.

2. Ada Code Inspection

Code inspections can be used by a large number of people on a software project and are applicable during the development, testing, and maintenance phases of a software project. Teaching code inspections implies references to programming style and standards. This project is using Ada programs as the work product for inspection. Thus, the code inspection simulation permits the introduction and discussion of Ada programming style and standards.

For the ALT prototype a single modestly sized piece of code was desired for the inspection product. In 1978 Glenford J. Myers of the IBM Systems Research Institute performed a controlled experiment in inspections [Myers 78]. Myers wrote a PL/I program based on an Algol program written using techniques of program-correctness proofs, by Naur [Naur 69] and in which six errors were discovered by Goodenough and Gerhart [Goodenough 75]. Myers translated the program to PL/I and introduced several more errors bringing the total to 15.

The PL/I program was translated into Ada for the ALT prototype. There are a number of stylistic points of discussion that have been introduced. This brings to almost 70 the number of points of discussion directly related to the documents being inspected.

In the final system there will be at least two Ada artifacts available for inspection. One will be a simple program using only a introductory subset of Ada, and the second will be a more complex program possibly using tasking and generics. When the principle intent of the course is to teach inspections, then the introductory subset can be used, even by audiences with no familiarity with Ada. Both artifacts can be used when instruction in Ada programming style, in addition to inspections, is to be taught to more advanced Ada practitioners.

3. Inspection Simulation and Expert System

The Advanced Learning Technologies Project is using a CD-ROM based interactive video technology as a hardware technology and expert system technology as a software technology. Experience has shown that reading or lecturing alone does not provide sufficient experience in inspections. The technology that can best provide the needed experience is a highly realistic interactive video simulation. This technology combines the visual and audio presentation of videotape with the interactivity and intelligent tutoring capabilities of computer-based systems.

Participants in inspections have well-defined roles. Typical inspections are composed of a moderator, a reader, a recorder or scribe, and often the producer of the code. To best simulate the interactions between participants in the inspection and to provide the ability for the user to take any role in the process, a rule-based expert system has been developed to model the participants. This expert system is used to define the "personalities" and to control the dialogue between the simulated members of the inspection team and the user.

The expert system developed for the prototype is composed of over one hundred rules. The rule base is used to make decisions in areas such as who should speak, the tone they should take, the content of what is to be spoken, and who should be addressed.

To model different personalities, several attributes are defined for each individual simulated by the system. Attributes include defensiveness, aggressiveness, talkativeness, and the tendency to make irrelevant humorous comments.

State variables keep track of the history of the conversation. These variables capture information on the current topic of discussion, length of focus on this topic, resolution of error topic, position on the topic for each participant, the current speaker, the person focused on in the current comment, and the person addressed in the current comment.

Rules control the conversation and model the personalities of the participants. For example, one rule deals with a participant who has been dominating the conversation.

This rule behaves as follows:

> One of the participants in the inspection has spoken too frequently during the last few minutes and someone (participant Y) other than this person is willing to talk.
>
> If the user is taking the role of the moderator then usually the user will get various types of feedback such as other participants beginning to lose interest or falling asleep. Frequently, participant Y will make a comment to the user about the problem.
>
> If the user is not the moderator (or occasionally when the user is the moderator) and if participant Y is fairly aggressive then Y tells the person talking too much to talk less. If the user is the moderator and this problem has been brought up to the user before then participant Y will more forcefully tell the user there is a problem (i.e. someone is dominating the conversation and the user, as moderator, should do something about it). If Y is not aggressive, Y will simply suggest that others talk more. This rule adjusts the propensity to talk of the individuals in the group. Other rules interact to alter the defensiveness of the person (user excluded) who is talking too much.

From the onset of the development of the prototype one of the largest questions related to user input. The highest fidelity simulation would require continuous speech recognition, a task not achievable in a workstation environment much less on a personal computer. At the other extreme the simulation would be to designed so that only simple multiple choice questions are used, an unacceptable alternative.

A text-based natural language interface is a solution falling in between the ideal and the unacceptable. The parsing of free form natural language sentences is still not performed well in anything but limited domains. Even then workstations with 10 megabytes of memory are used with 85-90 percent accuracy being the norm. Since our delivery environment is to be a PC-AT class machine, an alternative was sought.

Harry Tennant at Texas Instruments has done extensive work in a menu based natural language system: Natural Access. In Natural Access, sentences are constructed from sentence fragments. While allowing the user great freedom and power in sentence construction, this method restricts the domain and syntax to a level manageable by our target machine.

A similar interface was built for the Advanced Learning Technologies Project. The biggest problem, however, is the analysis of the sentences to be constructed. To help in this analysis a joint project with Elliot Soloway from Yale University was undertaken. Soloway had performed a significant number of inspection's analyses during a research project with IBM-FSD [Letovsky 88]. After discussions with Soloway, it was determined that a joint project would allow for the extension and tailoring of this research to a form that would provide the foundation for the ALT language interface subsystem.

With the expert system controlling the dialogue of the simulation and a partially constrained natural language interface, a high fidelity simulation is possible. This simulation provides the potential for real experience in inspections in less time and with increased productivity.

4. Effectiveness of Interactive Video

Reasons for choosing interactive video include cost and learning effectiveness. A typical classroom course has a modest, one-time development cost and high delivery cost. A single course may be given thousands of times at hundreds of locations over an average five year lifetime. Each time it is given an instructor presents the course, grades papers, and provides follow-up tutoring. By comparison, interactive video training has a higher development cost but much lower delivery cost. The only continuing delivery costs are management ones that are incurred to provide students access to the materials and assistance in turning on the equipment. The courseware presents the material, grades the student, and provides follow-up tutoring. Air Force studies which have considered all of these factors have shown interactive video courses to be 3 to 5 times less expensive than conventional classroom training [Advanced Technology 85].

Army studies have shown that students learn material more fully and in less time with interactive video, even when compared against computer-based training. In one study [Kimberlin 82], three groups of students took a course on missile electronic troubleshooting. One group took a standard lecture course, one a computer-based training course, and one an interactive video course. After the courses the students were given actual system faults. The lecture group solved 25% of the faults. Both the computer based training group and the interactive video group found 100% of the faults. While the classroom and computer-based training groups took about the same time to find the faults, the interactive video group found the faults in one-half the time of the other two groups. Studies in other domains have consistently found similar results [Bernd 83, Hon 83, Kimberlin 82, Pietri 87].

Transitioning the software engineering content is a fundamental goal of this project. However, none of interactive video's advantages would be helpful if the learning technology was not accepted. Fortunately, there is a growing installed base of interactive video in colleges, industry, and the services. Of all US organizations with 50 or more employees, 15.5 percent used interactive video to deliver job-related training in 1986, up from 11.6 percent in 1985. Thirty-six percent of all organizations of over 10,000 use interactive video [Training 86].

Presently the Army has 5,000 interactive videodisc systems installed. These are a mixture of several different systems for a variety of purposes including mapping, training, and archival data storage. Over the next four years the Army plans on purchasing 47,900 more units as part of the Electronic Information Delivery System (EIDS) project [USATSC 86].

The Air Force has 1,800 interactive videodiscs in use. Under development is the Air Force's Advanced Training System (ATS). ATS is a similar hardware system to the EIDS with the additional capability to interface with Compact Disc Read Only Memory (CD-ROM). (The Air Force Academy just bought 50 interactive videodisc systems and is ordering 1,000 CD-ROM players.) During peak usage, ATS will support the delivery and management of over 30,000,000 hours of technical training per year [Pohlman 87].

Interactive video technology has matured to the point where it is clearly a viable medium for the delivery of education and training. Videodiscs have been developed in project management, written and oral communications, physics, chemistry, mathematics, troubleshooting skills, and medical diagnosis [Stevens 87]. Videodiscs have also been developed for training in operating systems (UNIX) and computer languages (C). At present, no other projects deal with computer science or software engineering. Evidently the immaturity of the disciplines is the main reason for this as the medium is being used very successfully in other equally abstract areas.

5. Conclusions

The Advanced Learning Technology's interactive video simulation of Ada code inspections provides:

- Broad and comparatively inexpensive dissemination of the technique.
- Decreased time needed to introduce the new technique.
- Practical experience in the methodology.
- Continuous assistance in the techniques as they are being applied in the actual work environment.

Interactive learning technology holds the promise of high quality instructional design, inexpensive group-process simulation (allowing for self-paced, one-at-a-time training), and broad, relatively inexpensive dissemination. Ada code inspection methodology is a high leverage topic. The Advanced Learning Technologies Project is combining these two areas together in a unique, high leverage product for software engineering education and training.

References

[Advanced Technology 85]
Advanced Technology.
NonPersonal Studies and Analysis Services for Assessment of New Training Technologies.
Technical Report F41689-84-c-0012, U.S. Air Force, , June, 1985.

[Bernd 83] Bernd, R.
U.S. Army Training Applications of Interactive Videodisc.
In *Interactive Videodisc in Education and Training. Proceedings of the Fifth Annual Conference of Video Learning Systems.* Society for Applied Learning Technology, Arlington, VA, August, 1983.

[Goodenough 75] Goodenough, J. and Gerhart, S.
Toward a Theory of Test Data Selection.
IEEE Trans. Software Eng. SE-1(2):156-173, 1975.

[Hon 83] Hon, D.
The Promise of Interactive Video.
Performance & Instruction Journal 22(9):21-23, 1983.

[Kimberlin 82] Kimberlin, D.
U.S. Army Air Defense School Distributed Instructional System Project Evaluation.
In *Videodisc for Training and Simulation. Proceedings of the Fourth Annual conference on Video Learning System proceedings.* Society for Applied Learning Technology, , August, 1982.

[Letovsky 88] Letovsky, S., Pinto, J., Lampert, R.,and Soloway, E.
A Cognitive Analysis of A Code Inspection.
In Olson, Sheppard, Soloway (editor), *Empirical Studies of Programming.* Ablex Publishers, NJ, 1988.

[Myers 78] Myers, G.
A Controlled Experiment in Program Testing and Code Walkthrough/Inspections.
Communications of ACM 21(9):760-768, 1978.

[Naur 69] Naur, P.
Programming by Action Clusters.
BIT 9(3):250-258, 1969.

[NSB 86] NSB Task Committee On Undergraduate Science and Engineering Education.
Undergraduate Science, Mathematics and Engineering Education.
Technical Report NSB 86-100, National Science Board, Washington, D.C., March, 1986.

[Pietrasanta 87] Pietrasanta, A.
Keynote Adress.
In *Conference on Software Engineering Education.* Conference on Software Engineering Education, Pittsburgh, PA, April, 1987.

[Pietri 87] Pietri, Jr., J.
An IBM Perspective.
In *Optical Discs-An Information Revolution.* IEEE Conference on Optical Disc Technology, New York, NY, February, 1987.

[Pohlman 87] Pohlman, Lt. Col. David.
United States Air Force's Advanced Training System.
In *Proceedings of the 1987 Conference on Technology in Training and Education.* American Defense Preparedness Association, Colorado Springs, CO, March, 1987.

[Stevens 87] Stevens, S.
Interactive Videodisc, A Background Report.
Technical Report, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, 1987.

[Training 86] Training Magazine.
Annual Industry Report.
Market Report, Training Magazine, Minneapolis, MN, October, 1986.

[USATSC 86] USATSC.
Electronic Delivery System Primer.
Technical Report 41905-86, U.S. Army, , 1986.

This Page Left Blank Intentionally

INDEX

N O T E S

NOTES

N O T E S